INTERACTIVE TECHNIQUES FOR A PROGRAM GENERATOR USING PROLOG

by

K. S. Venkatesh

B.S., Mechanical Engg., Bangalore University, India, 1979

M.S., Mechanical Engg., Kansas State University, 1984

A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

William Hankley

Major Professor

TABLE OF CONTENT

List of Tables and Figures

Figures

----- *** -----

Tables

ACKNOWLEDGEMENTS

I would like to thank my major professor Dr. William J. Hankley for his valuable advice, guidance, and much needed encouragement throughout the course of the thesis work. I would also like to thank Dr. David Gustafson and Dr. Austin Melton for serving as committee members for the thesis examination.

Chapter I

OVERVIEW

1.1 Introduction

A program generator is a software tool that accepts some specification information from a user and creates what would be considered a source program in some high level language. Several different kinds of program generators are surveyed in Chapter II.

The work presented in this thesis extends concepts of two prototype generators previously developed at Kansas State University [Bart 85], [Pea 86]. Characteristics of these three generators are compared in Table 1.1. All three generators represent the knowledge of a class of programs as a BNF-like grammar with semantic actions which call for user selection and information input. The class of programs generated includes programs involving common algorithms for standard data structures such as tables, lists, and the like. Thus, these three generators synthesize library modules in some target language from algorithms stored in terms of grammar rules. In all these generators the external grammar was converted into some suitable internal form for efficient interpretation. The programs generated by all these prototype generators are in block-structured languages; [Bart 85] in Pascal, [Pea 86] in Modula II, and this thesis in a Pascal-like toy language. The dynamically generated programs are stored in some structured form, displayed,

Table 1.1. Comparison with other Prototype Program Generating Systems developed at KSU.

	1985 Barrett	1986 Peak	1987 Venkatesh
Implementation	IBM PC Pascal	VAX 11/780 C Prolog	IBM PC Turbo Prolog
Input Grammar	BNF-like with semantic actions (SA)	BNF-like with semantic actions (SA)	BNF-like with semantic actions (SA)
Domain	Library Module	Library Module	Library Module
Internal Representation	Tables - Definition - Token - SA Parameter Constant - Identifier	List + Rules	Facts - Definition - Token - Choice
Program Structure	Doubly Linked List	List Representation of Program Tree	Facts Representation of Program Tree
Target Language	Pascal	Modula II	Pascal-like Toy Language
Program Development	- Automatic Expansion - Display of Partial Program Traces	- Automatic Expansion - Final Program Display	- One Node at a time Expansion - Display of Partial Program Traces
Interactive Features	Limited	Limited	Not-so Limited

and saved in disk files. These three generators differ in the internal grammar representation, dynamically generated program structure, and interactive features during the program development process.

1.2 Contribution

The work by Barrett [Bart 85] demonstrated the fundamental concept of grammar representation of program domain knowledge, but it was an inflexible implementation. The work by Peak [Pea 86] showed the effectiveness of using Prolog for prototyping this kind of application which is based upon a top-down grammar rule expansion, but it also showed the limitation of using a pure functional style of Prolog programming (all dynamic structures are passed and returned by value). Generating even a small program using that generator could require a run-stack space of several megabytes. Both of those generators lacked effective user interaction. The user could supply specification information, but could not otherwise control the expansion.

This thesis contributes two major concepts to the program generators developed by [Bart 85] and [Pea 86]:

1. The internal representation of grammar rules and program structure in terms of smaller chunks and the use of a database to overcome run-stack overflow problem. This also aids in efficient program structure manipulation during interpretation.

2. The design of a suitable interpreter to incorporate several interactive features and control during the development target programs.

(1) Internal Representation

The internal representation of grammar rules developed in the present work allows efficient manipulation during the program development. This internal representation handles grammar rules in smaller chunks than in [Pea 86]. These are stored as facts in the Prolog database. During the interpretation phase, these facts are refined one at a time and the results of these refinements are dynamically stored as program facts simulating the program tree structure. Storing partial refinements as program facts in the Prolog database solves the storage problem of run-stack space during interpretation. The data structure of program facts in the Prolog database allows interactive program structure manipulation during program development. Thus, the internal representation of grammar rules and generated programs as Prolog facts simulating the tree structure enables interactive program developement with efficient memory usage.

(2) Interactive Program Development

The following principles [Tei 81] of user interaction are incorporated as specific features of the program generator which was implemented.

a. Specialization

Specialization is the process of refining non-terminals to their equivalent right-side terms. The parameterized algorithms are abstracted as grammar non-terminals in multiple layers. The layers should be conceptually clear to the user developing a program as successive refinements of grammar rules. This feature is incorporated in the present implementation by displaying the partially developed program traces with each grammar rule refinement

b. Constraint

The user's focus of attention is always restricted to objects on the screen which can be manipulated further. Such constraints are included in the present work as structured cursor movements which skip over the parts of program which cannot be modified. Modifiable components are highlighted as the cursor is moved to select them.

c. Consistency

All aspects of the user interface are based on a single concept of the program tree structure represented in terms of grammar rules. The cursor movements using arrow keys to the parent, child, left, and right non-terminal nodes in the present implementation are based on this principle.

d. Manual Control

Manual control is provided in all situations in which a user may choose an operation amongst many available. An example may involve choosing any non-terminal for further refinement. Manual

control enhances system flexibility. There are subtle problems though, such as trying to expand a non-terminal which requires previous declaration. Some semantic constraints should be imposed in such situations. Such semantic constraints are not implemented in the pressent work.

e. Immediate Visual Response

The display is always updated corresponding to any change in the data structure. The immediate response also included error reporting in cases such as wrong syntax (e.g., for an identifier name). These responses allow the user to monitor the state of program development process.

f. Multiple Conceptual Levels

In order to focus attention in and around some portion of a program, some details can be omitted from the display. This allows the presentation of the overall picture of the program in coarser detail and a particular portion in finer detail. This shrinking of details and re-displaying of the detailed version on command reflects the multiple views of the program being synthesized. The shrinking and re-display of details were implemented as ellipsis commands in the present work.

Note: Reversibility would have been another important feature, but was not implemented in the prototype. The reversibility command "undo" is desirable because it eliminates user anxiety about making mistakes.

The basic concepts of the present work have been gathered mainly from the following sources:

1. The paper by Warren [War 80] illustrates a methodology of logic programming for compiler writing. This idea has been employed in logic-programming-based program synthesis.

2. The work by Waters [Wat 82, 85] illustrates the usefulness of storing commonly used algorithms of standard data structure as program plans during program synthesis. The plans also incorporate the representation of data flow and control flow in the programs. This concept has been applied in storing the algorithms in terms of grammar rules. No attempt has been made to represent the data flow and control flow.

3. The paper by Teitelbaum [Tei 81] describes a structured editor environment, the Cornell Program Synthesizer, for program synthesis. This paper has motivated the use of structured representation and manipulation of program text during the synthesis phase.

4. The paper by Olsen [Ols 85] describes the organization of a user interface generation tool for editing templates. The internal data structure modifications and immediate display of changes requires close association of screen coordinates with the data structures. This paper has influenced the selection of proper data structures for efficient screen manipulation.

The present work is an attempt to integrate some of the key ideas presented in these papers into a unified framework which can serve as a basis for interactive program synthesis. It is

hoped that this basis will provide greater flexibility and user-friendliness during the development of target programs.

1.3 Implementation

A program generating system was developed to incorporate the concepts discussed in the key papers cited above. The system was written in Turbo Prolog [Tur 86] and implemented on an IBM PC compatible personal computer. The main issues were efficient space utilization and providing a good user interface in a microcomputer environment.

The programs generated by the system are in a toy language (block structured Pascal-like language). These generated programs are representative library modules which incorporate common algorithms for standard data structures such as tables and lists. The implementation of the system is merely a demonstration of feasibility.

Figure 1.1 describes the prototype program generating system. The program generator uses a translation grammar which encodes the program templates of the target language as its input (see Appendix II). The target language program templates are parameterized through semantic actions which include data type and associated algorithm selection and the user supplying identifier names for the variables.

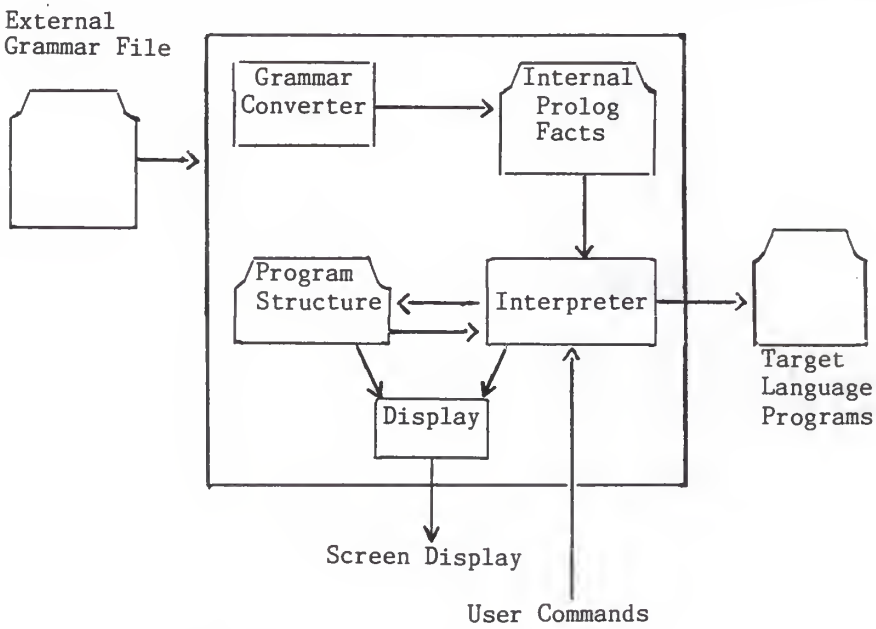


Figure 1.1. Program Generator System Overview

The main components of the system are the grammar converter and the interpreter. The grammar converter reads in an external grammar which is in BNF style and converts it into internal Prolog facts, one for each terminal, non-terminal and semantic action. The facts are of three types. A definition fact for each non-terminal links the left-side of a production to its right-side tokens. The token facts link each token in the right-side of productions. The choice facts provide a choice list for semantic actions in the grammar. These facts can be stored in external disk files and read into the Prolog database on demand. Since the grammar converter has not been implemented, the Prolog facts representing the external grammar are manually entered into the system.

The interpreter refines the grammar rules represented as Prolog facts with the help of a user. The interpretation starts with the fact representing the start-symbol which forms the root of the program structure. Refinement is done one node (fact) at a time. The interpreter takes a user command and current node as input, processes the current node according to the user provided command, and returns the next node to be processed. Refinement by expansion is the heart of the processing. The expansion of a non-terminal node links the corresponding right-side terms into the program structure. When semantic actions are executed the user is queried to provide an identifier name or to choose a grammar rule. The interpreter dynamically builds the program structure and identifier table. Program structure and

identifier tables are stored in the Prolog database as program and symbol-table facts. The refinement is complete when all the non-terminal nodes in the program structure are expanded. The leaf terminal nodes of the program structure constitutes the final program text.

Apart from the refinement command, interactive commands are provided for changing the current node through structured cursor movement, and for hiding and opening non-terminal nodes. This require a close association of screen coordinates with the program structure. The interpreter stores coordinates only for non-terminal nodes of the program structure. Updating screen coordinates along with program structure modification involves a significant amount of programming and forms the core of the interactive user-friendly interface. The reversibility command "undo" has not been implemented in the present work.

The user interface also includes the user dialogue interface with command menu display and pop-up menus for user selection with simple key strokes. Appendix I shows several screen dumps of actual terminal sessions during the development of a target program.

The continuous display of a partially developed program is handled by a display routine. This is a utility module called by the interpreter whenever the program structure is modified or there is a need for scrolling. The display module traverses the program tree and displays the program text (or a portion of program text) including the current node of interest.

The program generator is about 1450 lines of Turbo-Prolog code. The implementation requires 54K bytes of memory to store the compiler-generated form of the program source text. The run-time stack requirement was less than 16K bytes. The learning period for system development was about four months which included learning Prolog and program generator system concepts. The design and implementation of the system required about three months of effort.

1.4 Organization of the Thesis

Chapter II is a categorical survey of literature on program synthesis. The categories are deductive, very high level language (VHLL), transformational, programmer assistant and extended structured editor. The extended structured editor approach is dealt with in more detail as it directly pertains to the implementation developed in this thesis.

Chapter III identifies issues in the design of program generators. These issues include suitability of logic programming for program synthesis, space utilization, use of database to miminize the requirement of a large run-stack, internal representation techniques to store programming knowledge in the database and structured representation and manipulation of the generated program. A brief tutorial of the Prolog language is also included to show its usefulness for program synthesis.

Chapter IV describes the program generating system which was implemented. The description covers the design approach used to

aid good user interface and efficient memory usage, internal representation used to store the external grammar which parameterizes the stereotype algorithms for standard data structures, and the program structure of the final program generated. The questions faced during this implementation are typical for any program generator system. As a side effect, the necessity of employing a procedural style of programming in Prolog for efficient memory usage became evident. In addition, a number of lessons learned during implementation are shared.

The Appendix includes a sample input grammar, a hardcopy of program listing of the toy program generator, and a sample terminal session. The terminal session demonstrates the usefulness of a good user interface in the program generation process.

1.5 Discussion

The key issues of the present work can be summarized as follows.

a. Suitability of logic programming in Prolog for program synthesis

b. Space limitation considerations and use of database

c. Interactive features

The symbolic pattern matching and depth-first, left-to-right sequence of goal evaluation demonstrates the usefulness of Prolog programming for program synthesis using grammar rules.

The Prolog implementation for a program generating system will be small, modular and easily maintainable.

Space limitation is particularly severe for Prolog programs handling large data structures. For space efficiency the list structure should be broken into smaller facts stored in a database. This allows Prolog to handle larger programs with the limited space available. This method, however, requires extensive retrieval of facts from the database with modification and restoration. The final program will be generated as a side-effect.

From the very inception, Prolog and input/output have never blended properly. The built-in predicates offered by many Prolog systems for input/output are very minimal. However, the availability of Prolog compilers/interpreters on personal computers have overcome this deficiency to a large extent by providing a number of built-in input/output, window, and graphics predicates. This offers the possibility of exploiting the powerful pattern matching capability of Prolog along with an interactive user interface in the development of a program generating system. A close association of display update functions corresponding to the structural changes involves extensive programming to provide an interactive user interface.

It is clear that there are many desirable requirements which call for different system demands. Pattern matching is extremely suitable for Prolog programming. Space limitation requires a procedural style of programming with side effects (such as

assignments). A modular and efficient display structure for monitoring data structure changes suggests the use of access oriented programming with 'active values' [Bob 86]. A strong argument in favor of programming with multiple paradigm (Loops) and the inadequacy of any single programming language including Prolog for artificial intelligence programming is made in [Bob 85]. Arguments in favor of Prolog programming in the future for software development with the inclusion of many desirable modifications and additional features in the Prolog system is made in [Sub 85]. The availability of Prolog systems with many built-in predicates supporting interactive programming on cheap personal computers makes the development of program generating systems using Prolog a worthwhile effort.

Chapter II

AUTOMATED PROGRAM GENERATING SYSTEMS

2.1 Introduction

This chapter overviews four basic characteristics of program development systems: a specification method, a target language, a problem area (domain), and an approach or method of operation. Examples of ten major systems are described in section 2.6 to highlight the various characteristics and approaches taken in program generation. Table 2.1 summarizes these systems with respect to the four system characteristics.

2.2 Specification Methods

A specification method is a means for conveying to the generating system a description of the program which is required. Four kinds of specification methods are identified in the literature.

(1) Formal Specification

Formal specification methods are like very high level programming languages. In general, syntax and semantics of these methods are complete, that is, the specification completely and precisely defines the intent of the desired program. These specification methods are declarative in nature and convey the "what" of the program; the "how" program to be implemented is left unspecified. Most of the formal specification methods are

Table 2.1. Summary of Automatic Programming Systems

SYSTEM	Input Spec.	Output Lang.	Approach	Domain
PSI	Subset of Nat. Language	LISP	Transformation rules in KB.	Symbolic processing
CHI	High level Lang. V	LISP	Transformation rules in KB.	Symbolic processing, Graph thry.
PECOS	High level problem description	LISP	Transformation rules in KB.	Symbolic processing, Graph thry.
DEDALUS	Formal High Level Problem Description	LISP	Transformation rules in KB.	Numerical, Set Pgms.
SAFE/ TI	Preparsed Nat. Lang.	LISP	TI Transform rules in KB.	Scheduling, routing
PA	Pgm. text for analysis, Algorithmic description for pgm. synthesis	LISP/ ADA	Plans in KB.	Non numeric computing
CPS	Series of cmds. for program synthesis	PL/C	Grammar rules incorporated as procedures	Any problem
CPS-G	Lang. defn. in the form of attribute grammar	Any Block Struc. Lang.	Attribute gram. transformation	Structured editor
PSG	Lang. syntax, context reln., denotational semantics	Any Block struc. Lang.	Interpretation of rules of input language definition	Structured editor like environment
PECAN	Series of cmds. for program synthesis	Pascal	Hand-crafted procedures incorporating target lang. syntax and semantics	Any problem

not interactive, that is, the system does not interact with the user to obtain missing information, to verify hypotheses, or to point out inconsistencies. Good examples of high level languages used for formal specifications include SETL [Sch 81], V [Smi 85], and GIST [Fea 82]. These languages support higher level data structures such as sets, bags, etc., and the use of existential and universal quantifiers. The main issue in formal specification using high level languages is efficient implementation, i.e., efficient compilation.

(2) Specification by examples

Specification by example involves giving examples of what the desired program is to do. Sufficient examples allow these automatic programming systems to construct the desired program. The specification may consist of examples of the input/output behavior of the desired program, or it might consist of traces of how the program processes the input. The main issue in programming by example is that the specifications are rarely complete. To make the specification complete, a very large number of examples might be required which itself render such specification tedious and wasted effort. Illustration of programming by examples can be found in [Sum 77], [Bar 79], [Phi 77].

(3) Natural Language Specification

Natural language specification, probably the most desired specification technique, often lacks completeness because of the ambiguity in the natural language itself. The method of specifying in natural language involves interactive-checking of hypotheses, pointing out inconsistencies, and asking for further information. Examples of systems which acquire specifications through natural language include PSI [Gre 77], CHI [Smi 85], SAFE [Bal 76].

(4) Selection Menu

The user specifies the required program by selecting one of the many available programs catalogued in the system. This method provides a vehicle for unambiguous specification to the system. Selection from a menu is usually achieved by simple command key strokes. The specification is restricted to those available in the menu.

2.3 Target Language

Most program development systems generate programs in some target programming language such as Lisp, PL/1, GPSS etc. Some program development systems are versatile enough to develop programs in several target languages.

2.4 Problem Area

This is the area of intended application of the generated program. The problem area can be precise as in the case of NLPQ [Hei 74] which deals with simple queuing simulation problems. On the other hand, the application areas could be as diverse as I/O intensive data processing systems of Protosystem I [Rut 78] or symbolic computation (including list processing, searching and sorting, data storage and retrieval, and concept formation) as in PSI. The problem area plays a dominant role in the method of specification, the method of approach used by the program development system, and so forth.

2.5 Method of Operation (Approach)

The method of approach overlaps in many program generating systems. However, these systems can be broadly addressed in their method of approach such as theorem proving, program transformation, knowledge engineering, programmer assistant, and the extended grammar approach.

(1) Theorem Proving Approach

In this approach, the user specifies conditions that must hold for input data to the desired program and the conditions that the output data must satisfy. The conditions are usually specified in some formal language, often the predicate calculus. A theorem prover is used to prove that for all given inputs satisfying the input conditions, there exists an output that

satisfies the output condition. The proof yields the desired program as a side effect. Deductive synthesis is the approach used in such constructive program proving systems yielding the desired program with the required output assertions. A system which uses a deductive approach may construct programs incrementally during the proof process or in a separate post-proof phase. The basic problem in this methodology is that program proving is a difficult task and therefore no simplification is obtained during the program development process. Deductive program synthesis approach is employed by [Man 80], and [Der 85].

(2) Program Transformation Approach

The program transformation approach is probably the most widely used technique in program synthesis. Transformation refers to the process of converting a specification or description of a program into an equivalent description of the program. This is a vertical transformation, that is, a more abstract input (specification) is transformed into a less abstract executable source program. A lateral transformation can be used to work in the same level of abstraction but in a different perspective for achieving efficiency and removing redundancy. All these transformations are truth preserving.

Conventional language compilers are, in fact, transformational systems transforming a source language into machine interpretable code. However, a compiler differs from an

automatic programming system in that it applies transformations in a rigid, predetermined manner. In an automatic programming system, the application of transformations may depend on an analysis and exploration of results of applying various transformations. Systems like PECOS [Bar 79] and DEDALUS [Man 78] have a knowledge base with many transformation rules that convert parts of higher level descriptions into lower level descriptions, closer to the target language implementation. Such rules are repeatedly applied to parts of the program description with the goal of eventually producing descriptions within the target language. These systems develop a tree of possible descriptions of the program, with each descendent of a node being the result of a transformation. The goal of program synthesis in developing the tree is to find a description that is a target-language implementation of the desired program. The major issue in transformational systems is to control the application of transformation rules, in other words, to keep the transformation tree to a reasonable size. An excellent survey of research in program transformation is presented in [Par 83].

The transformation approach embodies the knowledge of program implementation in a library of transformation rules rather than in procedures. Thus, the implementation using this approach is modular and can be easily extended or modified.

Transformation systems may apply rules of transformation either automatically or under user control. Systems which are of limited power such as TAMPR [Boy 84] or PDS [Che 84] apply

transformation rules automatically by restricting the kind of transformation that can be defined and used. The PSI system is an example of a complex transformational implementation whose transformation module PECOS [Bar 79] works under the guidance of the efficiency module LIBRA [Kan 81].

The transformation approach closely associates with a knowledge-based approach in encoding programming knowledge as a set of transformational rules.

(3) Knowledge-Based Approach

A knowledge base is a database of facts about a domain, rules to manipulate the facts and other rules, and a control mechanism (inference engine) which controls the application of rules in a specific way. A knowledge base whose domain knowledge is "Programming" is indispensable in automatic program generation.

Building a knowledge base is a knowledge engineering task and is dependent on the specific goal for which it is built. The programming knowledge human experts possess is enormous and exists in different levels of granularity. Representing this knowledge in machine usable form requires that the various types of programming knowledge are fully understood. Precise representation of knowledge in an axiomatized mathematical way can only be applied to confined problem areas since programming knowledge is currently not very well understood. Barstow [Bar 79] argues that human programmers have collected such a wealth of

programming knowledge, that the only way to represent the knowledge for a large programming domain is through explicit rules.

The knowledge-based approach appears to encapsulate all other approaches discussed earlier. Formal specification and deductive synthesis approaches use knowledge representation in logic; the transformational approach uses knowledge representation in the form of rules and facts. The knowledge-based approach is getting attention because of it supports modularity; encoded knowledge (axioms,facts,rules..) can be added, deleted and changed. For generating a program for a given problem, the knowledge about the stored knowledge in the database (meta knowledge) can be used in deriving the applicability of the knowledge sources for a specific problem.

(4) Programmers Assistant

The basic concern of Programmer's Apprentice (PA) [Wat 82] system is program understanding and acting as a junior partner to the programmer. This approach is midway between an improved programming methodology and an automatic programming system. Program understanding might be defined as a system being able to talk about, analyze, modify, or write parts of the program. The intention of this approach is that the programmer will do the hard parts of design and implementation while PA will act as an assistant to the programmer in keeping track of the mundane details.

The understanding of programs in PA is through plans. A plan represents one particular way of viewing the program, or part of a program. A plan is a representation for a program which abstracts away from the inessential features of the program, and represents the basic logical properties of the algorithm explicitly. Matching the plan to a part of a program description corresponds to understanding the part in that way. Several plans can match the same part of a program, corresponding to different ways of understanding that part. Plans can also be built in hierarchical fashion.

(5) Extended Grammar Approach

The basic aim of this approach is to provide structured editing and multiple views of the program being developed. This approach provides only a supporting environment for program development rather than the complete synthesis of a program. The input to these systems is the textual program being developed. The editor in which the program is developed has the knowledge of the program structure, context sensitive relations and semantics of the target language constructs. The generation of these structured editors is mainly based upon the language definition in terms of the syntax, context relationships, and operational semantics of the language constructs. The language definition is based upon the extended grammar approach: encoding the context sensitive information in terms of attributes, and the meaning of language constructs in some form of denotational semantics. From

the language description, an editor is generated which stores the abstract tree as its primary view of program. The textual program input is converted internally into a concrete program tree. The correspondence of the abstract syntax to a concrete program tree allows various analyses including immediate recognition of syntax errors, some degree of semantic error checking, and structured editing of the program being developed. In addition, internally stored program fragments (or templates) allow program templates such as declaration, control constructs etc. to be introduced as program text with a few command key strokes. Thus, a program may be developed as a series of commands. The required details are filled in as textual input. This approach emphasizes the importance of an interactive environment during program generation. Lucid command menus, pop-up windows, use of mouse, etc. are common features of systems based on this approach.

The templates which are provided by systems based on the extended grammar approach are limited to language constructs which can be statically analyzed. The application domain of a program being developed using these systems can be anything. The programmer gets assistant-like help from these systems. In this respect, these systems are like the Programmer's Assistant (PA) system. However, PA also stores the commonly used program constructs (cliches) in the form of plans and hence is more powerful. Instead of control templates like WHILE or REPEAT which can be introduced as program text on command, a complete

sub-program, say a program for inserting an element to a list, could be introduced in PA. Thus, PA has a deeper and more dynamic understanding of a program being developed.

The advantage of a structured editor is that the approach is based on more formal grounds and hence it is relatively easier to develop a system by a complete target language definition in terms of a suitable grammar. The application domain is infinite and this limits the semantic analysis that a system can perform.

2.6 Example Systems

The example systems surveyed are knowledge-based translation rule guided systems and systems which provide structured-editor-like environments for program synthesis.

2.6.1 PSI system

The PSI system was developed by Cordell Green and his colleagues at Stanford University [Barr 82]. Even though the system was developed about six years ago, it is described in greater detail because PSI offers a comprehensive view of the overall effort required in automating the programming task.

The design goal of PSI was the integration of the more specialized methods of automatic programming in a total system. The system incorporates knowledge engineering, model acquisition, program synthesis, efficiency analysis and specification by examples, traces or natural language. Research is continuing at Kestrel Institute and a successor system, CHI has been

developed.

In PSI, a program is specified by means of an interactive dialogue which includes partial specifications by examples of input/output pairs or by traces. Through the specification, the user furnishes both a description of what the desired program is to do and an indication of the overall control structure of the program. The problem domain dealt with is symbolic computation, including list processing , searching and sorting, data storage and retrieval, and concept formation. The PSI system includes (see Fig. 2.1)

a. The PARSER/INTERPRETER Expert
b. The DIALOGUE MODERATOR Expert
c. The EXPLAINER Expert
d. The EXAMPLE/TRACE Expert
e. The TASK DOMAIN Expert
e. The PROGRAM MODULE BUILDER Expert
f. The CODING (PECOS) and EFFICIENCY (LIBRA) Expert.

The overall operation of PSI may be divided into two phases:

a) Acquisition of specification of the desired program
b) Synthesis of the program.

(a) The PARSER/INTERPRETER Expert

In the acquisition phase, the PARSER/INTERPRETER Expert first parses sentences and then interprets these phrases into less linguistic and more program oriented terms which are then stored in the program net. The expert has knowledge about data structures (sets, records, etc.), control structures (loops, conditionals, procedures, etc.) and some algorithmic ideas (set construction, quantification, etc.).

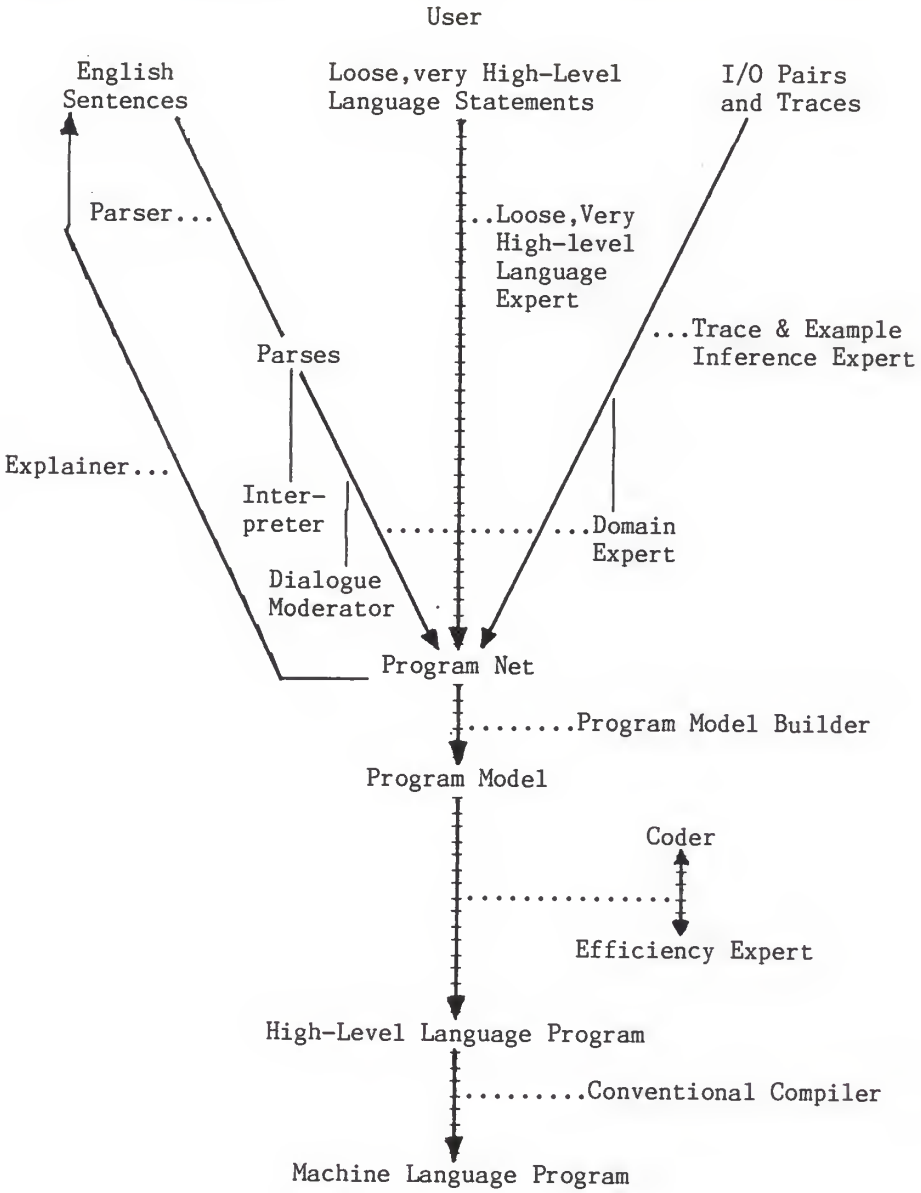


Figure 2.1. Major Paths of Information Flow in PSI.
([Barr 82], page 328)

(b) The DIALOGUE MODERATOR and EXPLAINER Expert

The DIALOGUE MODERATOR Expert poses questions to the users in order to guide the system in acquiring specifications of the desired program. The posed questions are in internal form based on relations and the EXPLAINER Expert converts the internal form into English descriptions which are presented to the user.

(c) The EXAMPLE/TRACE Expert

This expert handles simple loops and data structures from the specification by traces and examples.

(d) The TASK DOMAIN Expert

This expert uses knowledge of the application area to help the PARSER/INTERPRETER and EXAMPLE/TRACE Experts fill in the missing information in the program net.

(e) The PROGRAM MODULE BUILDER Expert (PMB)

This expert converts the program net into a complete and consistent program model. The PMB completes the model by filling in the various pieces of required information and by analyzing the model for consistency. Information is filled in by default, by the inference mechanism, or by queries to other experts, which may eventually result in a query to the user.

(f) The CODING (PECOS) and EFFICIENCY (LIBRA) Expert

These two experts are responsible for the synthesis phase. The CODING expert's knowledge base contains rules that transform parts of program descriptions into forms closer to the target language (LISP). The goal of the EFFICIENCY Expert is to guide the choice of the different rules, so that an efficient target language implementation eventually results. Because of the merits of the CODING Expert (PECOS) in program synthesis, it is described separately in a subsequent section.

2.6.2 CHI System [Smi 85]

The extension of the work on the PSI system has led to the design and implementation of the CHI knowledge-based synthesis system at Kestrel Institute. The goal of CHI is to provide not only a knowledge-based synthesis system, but also a supportive high level programming environment that includes specification acquisition, consistency checking, debugging, editing and maintenance. The CHI system uses a common knowledge base about the programming process to support all these activities.

The CHI knowledge-based programming environment emphasizes the use of a very high level language called "V" for specifying both programs and programming knowledge, and for interacting with the programming environment. The "V" language includes constructs for set mappings, relations, predicates, enumeration, program synthesis rules, and meta-rules for control. The high level description of knowledge in terms of the "V" language is

not only used for self compilation, but also for modifying and extending the environment itself.

The program synthesis is based on transformation refinement rules for handling data structure selection, enumeration constructs, and for producing concurrent programs from very high level program descriptions. These rules are also used for algorithmic design. Examples of algorithmic design include derivation of several prime finding and shortest path algorithms.

2.6.3 PECOS [Bar 79] and LIBRA [Kan 81]

PECOS serves as a coding expert of the PSI project. PECOS can also stand on its own and interact directly with the user. The problem area of PECOS is basically symbolic programming, which includes simple list processing, sorting, etc., and extended to include graph theory and simple number theory. Programs are specified in terms of very high level constructs such as data structure (collection, mapping, etc.) and operations (e.g., testing membership in a collection, computing the inverse image of an object under a mapping, etc.).

Knowledge about programming in the problem area has been made explicit and put into machine usable form, primarily as transformation rules, in PECOS' knowledge base. The system knowledge base consists of about 400 rules dealing with a variety of symbolic programming concepts. About 300 rules are general problem domain rules and the remainder are specific to the target language (LISP). The implementation techniques covered include

the representation of collections as linked lists, arrays, and boolean mappings, and the representation of mappings as tables, sets of pairs, property list markings and so forth. The transformational rules are internally represented as condition-action pairs. The condition parts of rules are partial configurations of abstract operations and data structures that are matched against parts of the developing program. When the match is successful, the rule action replaces parts of the abstract concepts with refinements of those parts. The complete program synthesis is obtained through gradual refinement, that is, repeatedly applying transformational rules which finally converts the initial abstract concepts into a concrete LISP program.

At some points during the transformational process, a conflict may arise because several rules apply to the same part of the program. Different conflict-resolution techniques ultimately result in different target language implementations (some times even dead ends) that often vary significantly in terms of efficiency. PECOS uses three methods for conflict resolution: user interaction, heuristic knowledge to choose the best rule, or if both these methods are not adequate for a conflict, apply all the rules in parallel.

In the PSI system, when PECOS works as a coding expert, choices between rules are made by an EFFICIENCY Expert called LIBRA. LIBRA incorporates more sophisticated analysis techniques, such as cost function, than the simple heuristics of

PECOS. PECOS does the synthesis part and LIBRA does the analysis part during program generation phase.

The success of PECOS demonstrates the viability of the knowledge-based approach in program generation. The efforts in developing this system have led to further research in encoding programming knowledge in different domains (which may result in convergence of transformation rules applicable to many problem domain [Bar 85]). Another research direction indicated by PECOS is the codification of different kinds of programming knowledge. Two types of knowledge seem particularly important: efficiency knowledge and strategic knowledge. LIBRA embodies efficiency knowledge to a limited extent. Much remains to be done in general strategic knowledge (such as divide and conquer) during program synthesis.

2.6.4 DEDALUS (DEDuctive ALgorithm Ur-Synthesizer) [Man 80]

This system accepts an unambiguous, logically complete, very high level specification of a desired program and through repeated application of transformation rules, seeks to reduce it on to an implementation in a simple LISP-like target language. This target language implementation is guaranteed to be correct and terminate. The programming knowledge is expressed via transformation rules. The rules which express general programming principles independent of specification and target language are of special importance. The DEDALUS knowledge base rules form conditional statements and recursive and non-recursive

procedure calls in addition to others.

DEDALUS is implemented in QLISP, an extension of INTERLISP which includes backtracking facilities. The domain of representative programs constructed by DEDALUS includes numerical programs (various GCD algorithms), and set programs (union, membership, cartesian product, etc.).

2.6.5 SAFE [Bal 78] /TI /GIST [Fea 82]

These systems were developed at the Information Sciences Institute (ISI) of the University of Southern California by a team headed by Robert Balzer. The SAFE system produces an automatic formal description in terms of functions from an informal problem description. The user of the SAFE system provides a behavioral description in a preparsed limited subset of English, including terms from the problem area. SAFE then seeks to determine a way of resolving all ambiguities and to fill in all missing information in a way that satisfies the system's knowledge of constraints that all programs must satisfy. The result is a complete, unambiguous, very high level program specification in a functional language called AP2. By employing its transformational rules, TI converts the functional specifications produced by SAFE into a high level algorithmic description in a specification language called GIST. The transformational rules in TI include UNFOLD, loop merging, rules for conditionals, and substitution of data structures with their representation. The transformed specifications in GIST are

further converted into a target program implementation.

2.6.6 PA (Programmer's Apprentice) [Wat 82], [Wat 85]

PA is a knowledge-based editor. As such, it lies between an aid to an improved programming methodology and a knowledge-based automatic programming system.

Programmers and the system work together during all phases of program development and maintenance. The programmer performs the difficult task of design and implementation, and PA acts as a junior partner and critic by keeping track of details and assisting in documentation, debugging and maintenance.

Program structures are represented as plans which are primitives or hierarchically composed of other plans. The basic entity of a plan is a segment defined by input expectations and output assertions. The relationship between plans is kept by defining links comprising data flow and control flow and semantic relations such as knowledge of how the behavior of a plan is inferred from the behavior of components.

The knowledge stored in PA is a database of common algorithms and data structure implementations called the plan library. PA's understanding of a program is embodied in a hierarchical plan for it. Typical plans include knowledge about the concept of a loop and its specializations into enumeration loops or search loops, or general techniques for manipulating trees, lists, arrays, and the like.

PA is based on an informal and flexible editing paradigm. Knowledge representation by plans allows both synthesis and analysis, and moving from abstract specifications to concrete programs, or vice versa. By providing a low-level plan structure from a given program, the system assists in analyzing already written programs. On the other hand, a user may construct a LISP (or ADA) program with the assistance from the system by naming a general algorithm and refining the abstract components into source code.

The translation approach in PA is not strictly truth preserving and there is no formal specification in the approach adopted by PA. Within the bounds of plan compatibility arbitrary changes to a program can be made. Thus, PA can be considered to be a transformational system only in a broad sense.

2.6.7 Cornell Program Synthesizer (CPS) [Tie 81]

CPS is a syntax directed editor developed at Cornell University. The entry and modifications of program text are guided by a grammar for the host programming language PL/C. The incorporation of the host language grammar into the editor guarantees syntactically correct programs and prevents syntactic errors on entry. The predefined language constructs are incorporated in the editor as templates. Programs are created top-down by inserting new templates and phrases in the skeleton of previously entered templates. Syntax error detection is immediate because template place holders can only be replaced by

syntactically correct insertions.

The programs are translated into interpretable form during editing and are then executed. Execution is suspended when an unexpanded placeholder is encountered and can be resumed only after the placeholder has been expanded.

The structured representation of the program (abstract syntax tree) allows structured cursor movements. This prevents unwanted cursor movement and quickens the program development process. The visual cues for template expansion and immediate visual response of edit and run time errors aid in quick and correct program development. The program is synthesized as a series of commands for inserting primitive program templates and specializing the templates according to the problem requirement.

Although CPS guarantees syntactically correct programs, it does not address semantics and algorithmic correctness at all. For this reason, CPS can only be considered to be a primitive program synthesizing system. Nevertheless, the structured representation of the program and the interactive features provided by this system improve the productivity of the programmers to a considerable extent. The problem domain can be anything, and hence CPS enjoys a wide usage.

CPS incorporates grammar rules as a set of procedures. These are distributed throughout the system. For this reason CPS is not modular and extension/modification requires substantial rework.

2.6.8 CPS-G (Cornell Program Synthesizer Generator) [Rep 84]

The synthesizer generator synthesizes a structured editor from an input language definition. The language definition is in the form of an attribute grammar which includes rules defining abstract syntax, attribution, display format, and concrete input syntax. From this specification, the generator creates a full screen editor for manipulating programs according to these rules.

In an editor generated with CPS-G, a program is represented as a consistently attributed derivation tree. Modification of a program corresponds to restructuring a derivation tree by pruning, grafting, and deriving. Incremental analysis is performed by updating attribute values throughout the tree in response to modifications. Such structured editors can be synthesized for any target language by a complete language definition of the languages in terms of attributed grammar.

Attribute propagation in the derivation tree is carried out by semantic equations. These semantic equations are part of the input grammar. Attributes can either be synthesized or inherited. Each semantic equation defines a value for a synthesized attribute of a left-side non-terminal or an inherited attribute of a right-side non-terminal. Context information is provided as an environment which is a set of identifier-binding pairs. The display of a program is defined by an unparsing scheme given for each grammar production. The display is generated by a pre-order traversal of the tree that interprets the unparsing scheme.

2.6.9 The PSG - Programming System Generator [Bah 85]

This system was developed at the Technical University of Darmstadt in West Germany. PSG generates sophisticated interactive programming environments from formal definitions of the target language. The formal language definition is a non-procedural definition of the language syntax, context conditions, and denotational semantics. The syntax is defined in BNF grammar style. The syntax definition includes a format definition, which is a tree-to-string transformation grammar. The format definition helps in constructing an external textual representation of an abstract tree. The syntax definition also includes definitions of headers and menu texts which are used to generate textual representations of templates and menus.

The context information is represented as "context relations" and a relational algebra defines context information for any particular node of the abstract tree. The context conditions are obtained by specifications of scope and visibility rules of the target language. The dynamic semantics of the language is defined in a denotational style. The semantics functions are defined as an extension of a type free lambda calculus. These semantic functions generate an interpreter for the language.

Parsing (building an abstract tree from textual input) and unparsing (textual output from an the abstract tree) are done incrementally to suit interactive development of target programs.

Using PSG, syntax directed editor-like environments have been generated from the language description of Algol60, Pascal, and MODULA II.

2.6.10 PECAN [Rei 85]

PECAN is a family of program development systems which support multiple views of user programs. These views can be representations of a program or the corresponding semantics. The primary program view is a syntax directed editor as in CPS. The semantic views include expression trees, data type diagrams, flow graphs, and symbol table. The system is implemented on Apollo workstations with a range of interactive features and graphic display capabilities.

The features provided by PECAN include

- immediate feedback of semantic and syntactic errors during program editing
- structured templates for building the program, available as commands
- the use of pop-up menus as alternative to typing most of the commands
- a multiple window display to make effective use of the screen
- incremental compiling.

Thus, in the PECAN environment, a program can be synthesized as a series of commands and with the details being filled up later. The semantic analysis is limited to structural primitives whose meanings can be derived statically. An algorithmic semantics is not included in the system. The system is hand-crafted and hence

is not modular. PECAN is similar to CPS, but has additional features for representing programs in many ways for the user. The interactive features of this system are far superior to the CPS system.

Chapter III

LOGIC PROGRAMMING FOR PROGRAM GENERATION

3.1 Introduction

This chapter outlines techniques for program gerneration using Prolog. Declarative and procedural semantics of the Prolog language are described and basic concepts of logic programming for top-down processing of a grammar are presented. The problem of space utilization in pure functional style logic programming is identified, and the alternative style using dynamic data structures in the database space is explained. Some methods for supporting user interaction with stored structures are also presented in this chapter. These methods form the basis for the implementation described in Chapter 4.

3.2 Prolog

Prolog is an implementation of predicate logic as a programming language. Prolog handles a subset of logic represented as "Horn clauses". Prolog is a declarative language. This means that, given the logic part of an algorithm in terms of facts and rules, the Prolog system will provide the control part. Prolog uses a fixed algorithm for evaluating goals and instantiating variables. This is sometimes referred to as "backward chaining"; it is equivalent to the top-down LL(1) parsing algorithm in compiler theory. The programmer essentially provides the 'what' of the algorithm in terms of logic and the

Prolog system provides the control component -'how' the logic is to be executed [War 80].

Clauses in Prolog are of the form:

Goal if SubGoal1 and SubGoal2 and .... SubGoaln.

There are two different ways of looking at the meaning of a Prolog program. In the declarative interpretation the Prolog clauses specify relationships between objects.

Example:

```
Concat([],L,L).  /* fact */
Concat([X|L1],L2,[X|L3]) :- Concat(L1,L2,L3). /* rule */
```

The declarative semantics of the above clauses can be read as:

> "The empty list concatenated with any list L is simply L. A non-empty list consisting of X followed by the remaining elements L1 concatenated with list L2 is the list consisting of X followed by remaining elements L3 where L1 concatenated with L2 is L3."

The alternate interpretation is obtained by considering the sequence of steps which is followed when the program is executed. The procedural semantics can be described as [War 80]:

> "To execute a goal, the system searches for the first clause whose head matches or unifies with the goal. If a match is found, the matching clause instance is then activated by executing in turn, from left to right each of the goals of its body (if any). If at any time the system fails to find a match for a goal, it backtracks, that is rejects the most recently activated clause undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal."

This procedural semantics can be explained with the "Concat" example. Consider the goal

```
Concat([1,2],[3,4],Z).
```

The result of the execution will be to substitute the required value of the variable Z. The goal matches only the second clause, and becomes

```
Concat([1,2],[3,4],[1|Z1]) :- Concat([2],[3,4],Z1).
```

The variable name 'Z1' is purely arbitrary. The process is repeated a second time giving rise to a further goal:

```
Concat([2],[3,4],[2|Z2]) :- Concat([],[3,4],Z2).
```

Finally Z2 gets its value from the first clause:

```
Concat([],[3,4],[3,4]).
```

Thus Z1 is [2|Z2] which evaluates to [2,3,4] and Z is [1|Z1] which evaluates to [1,2,3,4].

The above example illustrates the evaluation of concat(L1,L2,L3) with L1, L2 as inputs and L3 as output. If L1, L2, and L3 are all inputs, the clauses check for correctness. If only L3 is input, the program generates all possible combinations of values for L1 and L2 whose concatenation yields L3 by the built-in backtracking mechanism.

3.3 Applications using Grammars

The use of Prolog in compiler development is very appropriate because the system provides:

a. high level symbolic pattern matching of logic variables through unification
b. top-down left-to-right (depth first) application of clauses to evaluate the goal.

In particular, top-down recursive-descent parsing closely follows the Prolog system control mechanism. The clauses which are evaluated will be grammar rules of a target language and the

input is the program text in that language. Excellent illustrative examples are provided in [War 80], [Ste 86], and [Clo 82].

A grammar for a language is a set of rules for specifying the sequence of words (tokens) which are acceptable as a sentence in that language. Given a grammar for a language, any sequence of tokens could be examined to check whether it meets the criteria for being an acceptable sentence. This is done by establishing the underlying sentence structure. This is typically a parsing procedure in which the 'parse tree' of an input list of tokens is established.

Consider the example of an LL(1) grammar (left factorized and made deterministic to avoid unnecessary backtracking) as shown in Figure 3.1. The validity of the sentence '(a + b) * c' can be found by building a derivation tree as shown in figure 3.2.

Many Prolog systems allow the grammar rule notation '-->' which can parse a BNF-type grammar directly. The input to each rule is typically a list of tokens. Each rule consumes a part of the list (from left to right) and builds a structure corresponding to the consumed token. When parsing is successfully completed, the list left over should be the null list.

```
1.  E  --> T E1
2.  E1 --> + T E1
3.  E1 --> null
4.  T  --> F T1
5.  T1 --> * F T1
6.  T1 --> null
7.  F  --> ( E )
8.  F  --> a
9.  F  --> c
10. F  --> c
```

Figure 3.1. An LL1 Grammar for Simple Expressions.

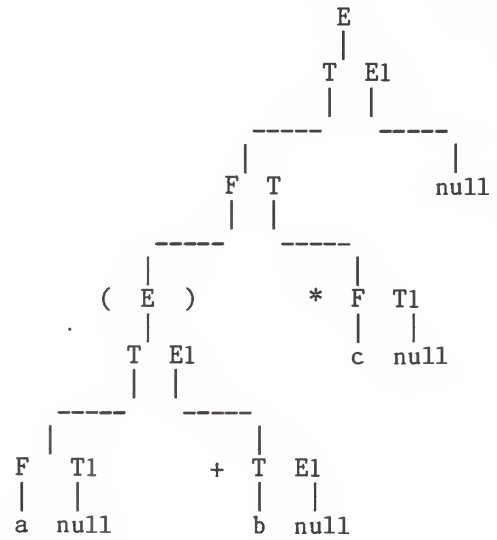


Figure 3.2. Derivation Tree for '(a + b) * c'

One such representation may look as follows:

```
E(InList,RestList,e(Structure)) :-
     T(InList,List1,Structure1),
     E1(List1,RestList,Structure2),
     Concat([Structure1],[Structure2],Structure).

T(InList,RestList,t(Structure)) :-
     ......

E1(InList,RestList,e1(Structure)):-
     ......
```

These rules build the structure as a list. For a given goal:

```
E([(,a,+,b,),*,c],_,ParseTree)
```

the clause returns the structure of ParseTree as shown in Figure 3.3.

The natural representation of grammar rules as Prolog rules makes parsing and building the derivation tree a simple task. The process of parsing can be summarized as shown in Figure 3.4.

Program generation in a restrictive domain can be visualized as the inverse of parsing, i.e., from a set of grammar rules, it is required to generate an instance of 'program text' in the form of a list through the application of grammar rules. From the example grammar (Fig. 3.1), each grammar rule can be written as follows:

```
E(Program) :-
     T(Pgm1),E1(Pgm2),
     Concat(Pgm1,Pgm2,Pgm).

T(Program) :-
     .....

E1(Program) :-
     .....
```

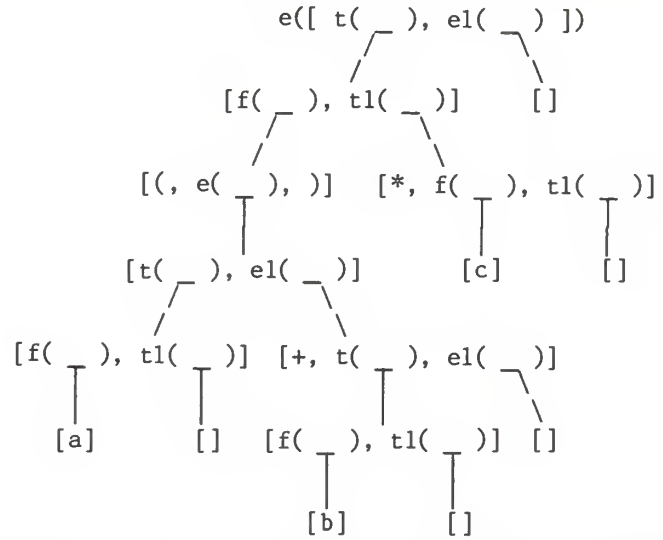


Figure 3.3. Parse Tree as a Structured List for '(a+b)*c'.

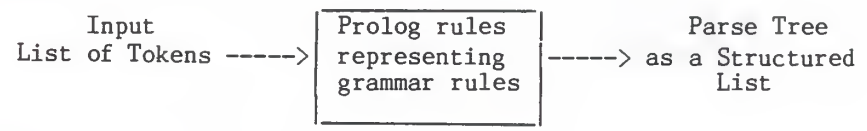


Figure 3.4. A Parser in Prolog

The components of Program which are Pgm1 and Pgm2 get bound to a value by subsequent applications of grammar rules to terminal token values. The values are propagated to the head of each clause (rule) and thereby Program is synthesized. All the work is done by the Prolog unification and control mechanisms. The synthesized sub-units may be simple lists, in which case the final program obtained will simply be a list of tokens constituting a valid program according to the grammar rules. Additional structuring can be imposed to get a trace of the derivation. This trace is a 'program tree' as shown in figure 3.3.

The following is an example of a set of rules for obtaining a program tree:

```
E(e(Pgm)) :-
     T(Pgm1),E1(Pgm2),
     Concat([Pgm1],[Pgm2],Pgm).

T(t(Pgm)) :-
     .....

E1(e1(Pgm)) :-
     .....
```

Unlike parsing, program synthesis requires further guidance in choosing one of the grammar alternation rules during top-down expansion. An interpreter can be visualized as providing functionalities such as guiding the application of grammar rules, obtaining values of identifiers from users, etc. Additional functions such as manipulation of the program tree being synthesized can be provided by the interpreter. Therefore, the

interpreter can process many other commands in addition to the expansion of non-terminals in the grammar rules. One such interpreter can be relationally represented as follows:

```
Interpreter(Command,InPgmList,ModifiedPgmList).
```

Interpreter modifies the 'InPgmList' according to 'Command' to get 'ModifiedPgmList'. If 'Command' has the value 'expand' (which is the central function of the interpreter), the 'InPgmList' could be a list of one element, viz. the start symbol. For the example grammar of Figure 3.1, an interpreter might be invoked for expansion as follows:

```
Interpreter(expand,[e(_)],Pgm).
```

The expansion action starts from the leftmost item of 'InPgmList'. Any terminal in the list is left unchanged. A non-terminal represented as a functor of a compound-term gets a value for its term bound to the right-side terms of the grammar rule. For example, after one unfolding of a grammar rule, e(_) will become e([t(_),e1(_)]). All non-terminals can be expanded in a depth-first sequence, e.g., expanding t(_) next in the above example by calling the interpreter recursively with appropriate 'InPgmList'. When all non-terminals are expanded, the recursive calling stops and unwinds from the recursion. The value for the program structure is obtained at each recursion unwinding to synthesize the final value of 'Pgm'. The module development system developed by [Pea 86] was based on this approach.

3.4 Program Generation using the Prolog Database.

For a large program structure, the process of program generation described in the preceding section requires a large amount of run-stack space because of the recursive calls to the interpreter. Furthermore, this recursive programming leads to complete expansion of all grammar rules before the final program is returned. Although the program structure is maintained during refinements, the structured operations on the program tree can be incorporated only after a complete program is developed. Therefore, this approach is unsuitable for interactive generation of programs of reasonable size.

In order to overcome the run-stack space problem, the program structure should be handled as small chunks. Each non-terminal and terminal of grammar rules and program structure may be represented as individual facts in the Prolog database. The expansion of grammar rules can be done one fact at a time in the following manner:

```
Interpreter(expand,Node,NewNode) :-
   isNt(Node),!,  /* "Node" is a non-terminal */
   linkRHS(Node),/*graft RHS terms of "Node"topgm tree */
   getNextNode(NewNode), GetCmd(Cmd),!, /* return next node
          to be expanded "NewNode" and next command "Cmd" */
   Interpreter(Cmd,NewNode,NextNode).    /* recursive call */
```

The interpreter begins with the start symbol as "Node"; this is asserted in the Prolog database as a fact of the program structure. For expanding non-terminals, the interpreter retracts the facts corresponding to non-terminals, and stores the right-side of the non-terminal (corresponding to a grammar rule) as

facts of the program tree ("linkRHS(Node)"). At any given time, the Prolog interpreter will be handling only one node. This style of Prolog programming with extensive use of the system database and corresponds to programming based on side effects. The fracturing of program structure into smaller nodes aids in efficient memory utilization.

Turbo-Prolog is a statically typed language which is compiled before execution. The static features of Turbo-Prolog require that the facts which need dynamic modifications should be declared separately as database facts. Turbo-Prolog allows only those facts to be retracted, changed, and reasserted during execution.

The representation of terms in grammar rules (non-terminals and terminals) as nodes (facts) involves the additional effort of explicitly linking each node. The natural sequencing obtained from the list structure is no longer available. These links (such as parent and sibling node link) can be obtained by additional terms in the facts representing the nodes. Each fact representing a node is like a variant record in a procedural language. The terms representing the relationship between nodes correspond to the pointer fields in the records of a procedural language. The representation of grammar rules and program structure in terms of nodes and explicit links and the use of this representation for program synthesis using a procedural language interpreter can be found in [Bart 85].

Data structure representation in terms of small units such as nodes and explicit links requires procedurally oriented data manipulation. An excellent summary of various techniques for procedurally oriented programming in Prolog can be found in [Mun 86]. A detailed description of grammar and program tree representation in terms of facts in a Prolog database can be found in Chapter IV of this thesis.

3.5 Interactive Loops in Prolog

An interpreter working interactively on input commands can be devised as a tail recursive loop:

```
Interpreter(quit,_,_) :- !.
Interpreter(Cmd,Args,NewArgs) :-
     Read(Cmd),ProcessCmd(Cmd,Args,NewArgs),
     !,Interpreter(Cmd,NewArgs,NArgs).
```

Note the special built-in predicate ! (read as cut). This predicate prevents backtracking. The operational semantics of cut can be described as follows [Ste 86]:

> The goal (cut predicate) succeeds and commits Prolog to all choices made since the parent goal was unified with the head of the clause the cut occurs in. Thus, cut prunes all the alternative clauses below and conjunctive clauses to the left of the cut.

In the interactive loop, "Cmd" is read from the terminal, processed on "Args" to get "NewArgs", and the read/process cycle is repeated recursively. As described in previous sections, implementations based on recursive loops require larger memory spaces. Although many Prolog systems incorporate tail recursive

optimization, interactive loops can be represented as "failure driven" loops which guarantee a constant system memory requirement.

```
Interpreter :-
     Repeat,Read(Cmd),ProcessCmd(Cmd,Args,NewArgs),
     Cmd="quit", !.
Repeat.
Repeat :- Repeat.
```

The goal, Repeat, is always true. When "Cmd" is not "quit", the failure invokes the system backtracking mechanism by which a new "Cmd" is read. Note that the predicate "ProcessCmd" must be deterministic for every "Cmd" (i.e., no alternatives for a given "Cmd"). When the command is "quit", the cut succeeds and the repetition is stopped.

3.6 Display Update

Program generation from a given set of rules is obtained by successive refinement of non-terminals to their right-side terms. Commencing from the start symbol, which will be the root of the program tree, refinements build a concrete program tree. Additional functions of the system interpreter act on the underlying program tree and update/modify the tree. The challenge in the design of an interactive user interface is the immediate update of the program textual display corresponding to the structure modification.

There are two aspects of an interactive interface: input dialogue interaction with the user, and display update. Input

interaction with the user is relatively easy as compared to display update. The input dialogue interaction provides command menus, designated keys for predefined commands, help menu display, etc. The designated keys for commands simplify command entry and eliminate the necessity of memorizing a complex command syntax. The display update traverses the data structure and displays it on the screen. In order to associate a component of a data structure with a location on the screen, either the position can be calculated on demand or stored in the data structure as an attribute. Storing the screen location has the advantage of quicker response to commands, such as cursor movement, which do not modify the underlying structure. On the other hand, structure update commands have to do additional work in storing the screen locations along with other processing such as the expansion of a tree node. As a compromise, the screen location can be stored for those nodes which can be modified such as non-terminal nodes in the program tree.

A sophisticated "User Interface Management System" (UIMS) for editing templates of programs has been suggested in [Ols 86]. A simple approach is presented in Figure 3.5 which is suitable for synthesizing small programs.

In this implementation, the screen is updated by refreshing the entire screen. This method can be slow if the program generated is large. However, for small modules this approach will be satisfactory and is easy to implement.

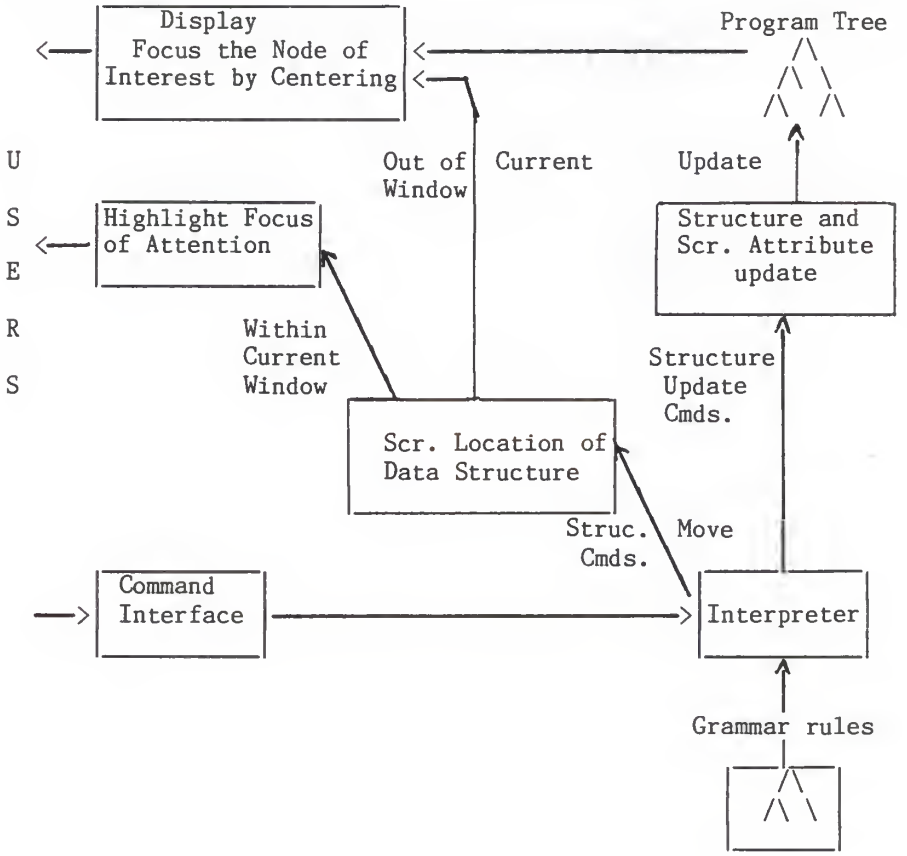


Figure 3.5. System Configuration for Interactive Program Synthesis

Chapter IV

IMPLEMENTATION OF THE PROTOTYPE PROGRAM GENERATOR

4.1 Introduction

This chapter describes the program generator in considerable detail. First, the external grammar input to the system is described. Next, the data structures of the internal representation of the grammar rules and the program tree structure are explained. Finally, algorithms of important predicates used in the system are presented.

Ideally, the program knowledge input should be in the form of a BNF style translation grammar. The grammar will be converted into a suitable internal form on which the system interpreter operates. The grammar conversion module has not been implemented in this system, but the algorithms required by the converter are included in this chapter.

4.2 External Grammar

The external grammar which is in BNF style encodes program templates of the target language. The templates are parameterized through semantic actions. This grammar is internally represented as Prolog facts. The grammar syntax is as follows:

1. The top production must be the start symbol.
2. The left-side of all grammar production must be a unique non-terminal. The non-terminals are enclosed in angle brackets '<' and '>' and are separated from the left-side terms by

'-->'. The uniqueness of the left-side non-terminals helps unambiguous grammar rule application by the interpreter. The alternation in the grammar must be rewritten as a separate non-terminal deriving a semantic action "choose" to select one of the alternation rules.

3. The terminals in the right-side of production rule appears in single quotes. E.g., 'Program'.

4. The non-terminals in the right-side are enclosed in angle brackets '<' and '>'. E.g., <dcls>.

5. The format statements used for pretty printing that indicate new line and amount of absolute indentation is represented as nl(tab). E.g., nl(3).

6. Semantic actions are delimited by dots. Eg. .sa(X).

7. The end of production is indicated by the slash '/'.

The absolute indentation of the format statements simplifies the implementation of the "display" routine and the program structure update functions which record the co-ordinates of program nodes. Usage of absolute indentation is possible only because operations such as "cutting" and "pasting" of program node sub-trees during program generation are not permitted in the implementation of the interpreter.

Semantic action representation is performed in a special way. A non-terminal deriving a semantic action as its right-side term will not have any other term. In other words, semantic actions will be the only terms in the right-side of grammar rules. This restriction simplifies the translation and execution of semantic actions during interpretation.

Figure 4.1 shows the example input grammar. The grammar encodes the table and the list algorithms of a toy language. An

```
<pgm> --> 'Prog' <pgmid> nl(3) <dcls> nl(0) 'end' <pgmid> /

<pgmid> --> .id(pgmid). /

<dcls> --> .choose(abstractty,[table,list]). /

<table> --> <tabletydef>  nl(0)  nl(3)  <tableprocs>  nl(3)
            <moretableprocs> /

<tabletydef> --> 'type table' /

<tableprocs> --> .choose(tableprocs,[tableinit,tablesort]). /
      .
      .
      .
<moretableprocs> --> .more(tableprocs). /
      .
      .
      .
```

Figure 4.1. Representation of External Grammar.

alternation rule in the grammar is implemented by the CHOOSE semantic action. The CHOOSE semantic action provides a list of alternate grammar rules for a particular type of choice. The execution of choice SA (Semantic Action) results in the selection of one of the grammar rules by the user. This provides a way of guiding translation rules during interpretation. A repetition loop of a grammar rule is encoded by MORE SA. This SA is executed only on user demand. The execution of MORE SA allows application of a grammar rule several times.

4.3 Internal Grammar Representation

The internal representation involves three types of Prolog facts: "definition" facts corresponding to the left-side of a grammar rule, "token" facts corresponding to the right-side terms, and "choice" facts for all alternations in grammar rules. Examples of these three types of facts are shown below.

```
Definition fact:
     d(Non_terminal_name, Right_side_index).
     E.g., d(pgm,1).

Token fact:
     t(Token_index, Token_node, Sibling_index).
     Token-node: nt(Non_terminal_name); E.g., t(4,nt(dcls),5).
                 const(Terminal);       E.g., t(1,const("Prog"),2).
                 nl(Tab)                E.g., t(3,nl(3),4).
                 sa(SAname, SA_Parm_type).
     SAname: getid;  E.g., t(8,sa(getid,pgmid),0).
             choose; E.g., t(9,sa(choose,abstractty),0).
             more;   E.g., t(17,sa(more,tableprocs),0).

Choice fact:
     c(ChoiceType,List_of_choices).
     E.g., c(abstractty,[table,list]).
```

The Right_side_index in a definition fact points to the first right-side token of a grammar rule. The Sibling_index in the token fact links the right-side tokens of grammar rules. The end of a grammar production rule is denoted by null(0) Sibling-index of the last token. The choice facts provide a link between CHOOSE SA token fact and a list of grammar rule choices. Storing the grammar rule choices as separate facts enables pop-up menu presentation of choices by the interpreter. This makes the system more user friendly. The internal form of the grammar rules are shown in Figure 4.2.

The translation of grammar to the internal form is easily automated. This involves get_right_side and get_left_side terms, and explicitly linking them together. The following algorithms show how this can be implemented.

Top level algorithm:

```
Loop till EOF
  Get_left_side term,
  Get_right_side terms,
end Loop.
```

```
Get_left_side
  read_non_term(ntname),
  assert d(ntname,TokNdx),
end Get_left_side.
```

TokNdx is a unique global index which provides an index to token facts and provides an explicit link between definition and token facts.

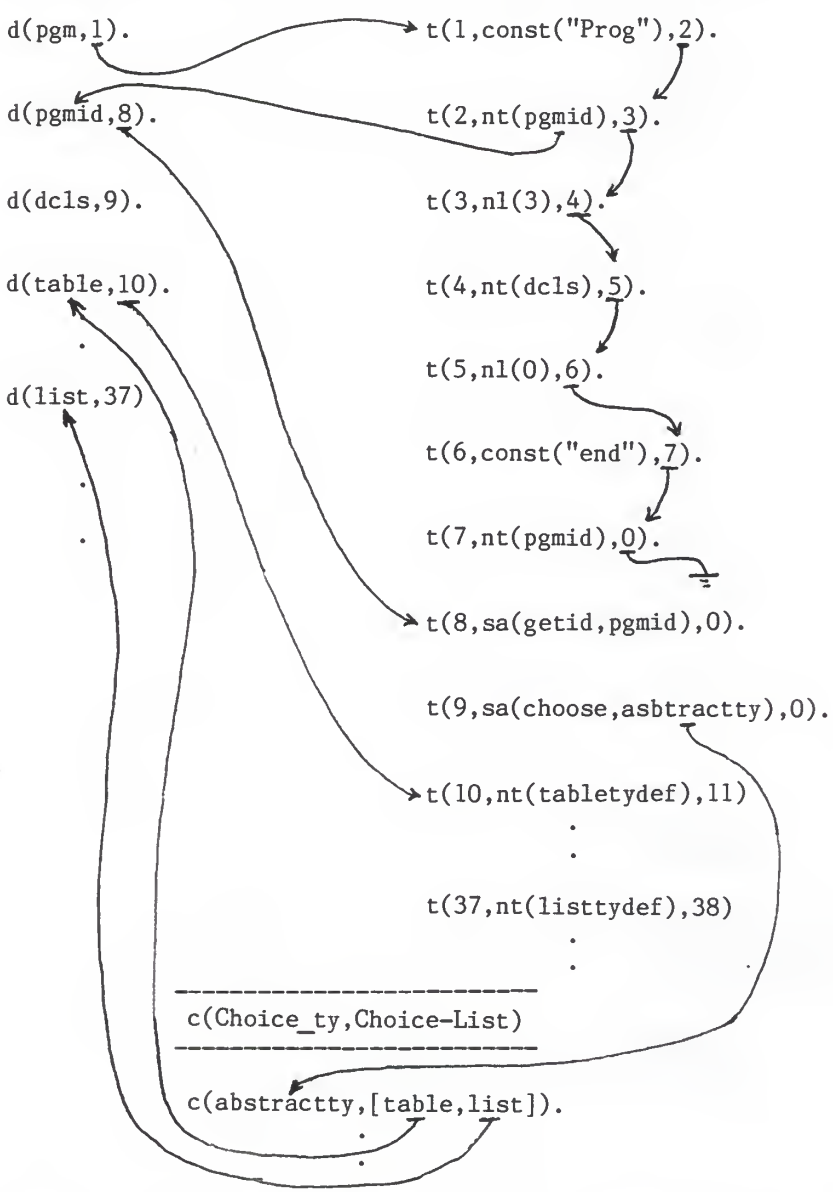


Figure 4.2. Internal Representation of Grammar Rules as Prolog facts.

Get_right_side terms:

This is more complicated as the tokens may be terminals, non-terminals, SA, and format tokens.

```
Get_right_side terms
  Loop
    read token,
    case token of
      SA
      delimiter:read Sa_name,
                case Sa_name of
                  choose:read Choice_ty, Choice_List,
                         assert
                           t(TokNdx,sa(choose,Choice_ty),0),
                           c(Choice_ty,Choice_List);
                  more:read More_ty,
                       assert
                         t(TokNdx,sa(more,More_ty),0);
                  id:read Id_ty,
                     assert
                       t(TokNdx,sa(getid,Id_ty),0);
                end case
                update TokNdx;

      terminal,
      non_term,
      format   :SiblingNdx is TokNdx + 1 ,
                case token of
                  terminal:read Const_name,
                           assert
                             t(TokNdx,const(Const_name),
                               SiblingNdx);
                  non-term:read Ntname,
                           assert
                             t(TokNdx,nt(Ntname),
                               SiblingNdx);
                  format:read Tab,
                         assert
                           t(Tokndx,nl(Tab),SiblingNdx);
                end case
                TokNdx is SiblingNdx;
      End of
      Production:Set the previously asserted last token-fact
                   SiblingNdx to 0,
                 Decrement TokNdx,
                 Exit.
    end case
  end Loop
end Get_right_side terms.
```

The global variables (TokNdx and SiblingNdx) can be implemented as Prolog database facts. The values of these variables can be changed by retrieving the fact and re-asserting back with a new value. This is like an assignment statement in a procedural language.

The grammar conversion algorithm is described in a fair amount of detail to guide the implementation of a future grammar converter as an extension to the program generator. The availability of the built-in Turbo-Prolog predicate "fronttoken" enables a straight-forward implementation of the algorithm.

4.4 Interpreter

The interpreter operates on the internally stored grammar rules in generating a program structure. The main functions of the interpreter are to parse through the grammar rules starting from the start symbol one node at a time and then to link the right-side terms of non-terminals in creating the program structure.

It is necessary to understand the program structure represented in the form of Prolog facts in order to understand the various functions that the interpreter provides.

Programs which are generated dynamically during interpretation are maintained as tree structures. A tree structure is formed by explicitly linking "program" facts which get asserted dynamically. A program tree node (fact) is represented as follows.

```
p(ProgNdx,Node,ParentLink,SiblingLink).

  ProgNdx: unique number for each program node.
  Node: const(ConstName); for terminals
        nl(Tab); for format nodes
        nt(Ntname,ChildLink,Visible,Expanded,Row,Col)
          ChildNdx: Program node index pointing to the first
                    RHS term or non-terminal grammar rule.
          Visible : Boolean flag (y/n) which is used by
                    "display" function to show or hide the
                    subtree corresponding to the non-terminal.
          Expanded: Boolean flag (y/n) to indicate wheather a
                    non-terminal node is expanded or not.
          Row, Col: Screen position of a non-terminal with
                    reference to (0,0) start location.
                    Different parts of the program tree can be
                    displayed by changing the reference from
                    zeroth row to any other row. This forms
                    the basis of scrolling.
```

It is clear from the description of program facts that the representation is geared towards interactive program manipulation and display.

The interpreter also generates one more data structure in order to keep track of identifier names provided by the user in a symbol table of facts

```
s(Idtype,Value).
```

Note that the scope information is not included and identifiers are global to the entire program.

4.4.1 Interpretation

The logic of the interpreter module is described in terms of algorithms which cover the important predicates used in the system. For a detailed presentation of the interpreter logic refer to Appendix III at the end of this thesis.

The interpreter is invoked by a top level predicate "go" which reads in the Prolog facts (representing the grammar) from an external file, asserts the start symbol as a "program" fact, initializes the screen location, sets up a command help menu, and calls the interpreter. The algorithm for "go" is presented below.

```
go()
  consult(Prolog_grammar_fact_file),
  Init(ProgNdx), Init(Screen_location),
  assert Start_symbol,
  set_up(Command_help_window),
  display(in:ProgNdx), /* display start index */
  interp,
end go.
```

The interpreter ("interp") reads and processes user commands in a failure driven loop. The following text describes the "interp" algorithm.

```
interp()
  Repeat
    read Cmd,
    Process_cmd(in:Cmd,in:Pnode,out:NewPnode),
  until Cmd='q',
  Save_Pgm(),
end interp.
```

The interpreter calls "Process_cmd" and passes it the user "Cmd" and the current program node to be processed. The interpreter loops until the user types 'q' to quit the system. The program text generated is saved in a predefined disk file by the interpreter before quitting as a precautionary measure.

4.4.2 Command Processing

The commands which are processed by the "Process_cmd" can be described as follows:

```
Process_cmd(in:Cmd,in:Pnode,out:NewPnode)
  case Cmd of
    Structure_move:Struc_move(in:Cmd,in:Pnode,out:NewPnode);
    (i.e., arrows)
    ellipsis ('.'):hide_children(in:Pnode,out:NewPnode);
    open_ellipsis ('o'):open_children(in:Pnode,out:NewPnode);
    expand_node ('e'):expand(in:Pnode,Out:NewPnode);
    save_pgm ('s'):save_pgm;
  end case
end Process_cmd.
```

4.4.3 Structured Cursor Movement

The "Struc_move" predicate allows movement to the child, parent, left, or right non-terminal nodes of a program tree. The current node of interest is highlighted by an inverse video display.

```
    .
Struc_move(in:Cmd,in:Pnode,out:NewPnode)
  case Cmd of
    uparrow   : move_out(in:Pnode,out:NewPnode);
    downarrow : move_in(in:Pnode,out:NewPnode);
    leftarrow : move_left(in:Pnode,out:NewPnode);
    rightarrow: move_right(in:Pnode,out:NewPnode);
  end case
end Struc_move.
```

The predicates move_in, move_out, move_left, and move_right invoke other predicates that move to the child, parent, left, and right non-terminal nodes respectively. The algorithms for all these movement predicates are similar.

```
move_in,out,left,right(in:Pnode,out:NewPnode);
  get_child,parent,left,right
                    non_term_node(in:Pnode,out:NewPnode),
  Chk_display(in:NewPnode),
end move.
```

The check display ("Chk_display") predicate checks whether the new program node ("NewPnode") co-ordinates are within the current screen window or whether there is a need for scrolling. If "NewPnode" is within the current window then the cursor (inverse video bar) is changed from "Pnode" to "NewPnode". Otherwise, a display procedure is called ("display(in:NewPnode)") to display the program text containing "NewPnode".

4.4.4 Ellipsis and Open_Ellipsis

Open and hide_children predicates operate on the flag "Visible" of the program node for non-terminals. This flag is turned on or off (open or hide) for the current program non-terminal node. The co-ordinates for all other non-terminal program nodes to the right of the tree from the current non-terminal node need to be updated ("modify_righttreeRC(in:Pnode, in:RowDiff,in:ColDiff)") and the program text is then re-displayed.

```
open(hide)_children (in:Pnode,out:Pnode)
  turn_on(off) the display flag for Pnode,
  calculate the co-ordinate diff. RowDiff, ColDiff,
  modify_righttreeRC(in:Pnode,in:RowDiff,in:ColDiff),
  display(in:Pnode),
end open(hide)_children.
```

4.4.5 Expand

The expansion ("expand(in:Pnode,out:NewPnode)") is the heart of the processing. This command grafts the right-side terms corresponding to the current non-terminal node ("Pnode") into the program tree with "Pnode" as the root and the right-hand terms as the branches of the tree. The expansion refines one node at a time and returns the next node ("NewPnode") to be refined. The selection of "NewPnode" for further expansion is based on a depth-first sequence. The expand was implemented as a single-step function rather than as an automatic recursive loop in order to increase the flexibility and manual control during expansion. The algorithm for "expand" is described below.

```
expand(in:Pnode,out:NewPnode)
  case Pnode_type of
    constant,
    format,
    Pnode already
      expanded      : NewPnode = Pnode;

    NtNode:Get RHS Token Node "TNode",
           assert "Pnode" as Ntnode,expanded,and not visible ,
           case Tnode_type of
             SAtype:Semact(in:Tnode,in:Num,in:Pnode),
                    chk_more_choose(in:Tnode,in:Pnode,
                                                out:NewPnode);
             others:Link_rhs(in:Tnode,in:Num,in:Pnode),
                    compute the difference "RowDiff" and
                      "ColDiff" of screen co-ordinates,
                    modify_righttreeRC(in:Pnode,in:RowDiff,
                                                  in:ColDiff),
                    expand_next(in:Pnode,out:NextNode),
                    chk_expand(in:Pnode,in:NextNode,
                                             out:NewPnode);
           end case
  end case
end expand.
```

The "expand" skips over terminals, already refined non-terminal nodes, and format statements. For a non_terminal program node ("Pnode") which is not expanded, the token_node ("Tnode") is first examined to check if any semantic action needs to be performed. The semantic action ("semact(in:Tnode,in:Num, in:Pnode)") takes the token and program node as well as the current index ("Num") which is used for explicit linking as its input. The execution of a semantic action results in the system asking the user to choose a production or to furnish an identifier value. The results of a "semact" execution are stored in the program and/or symbol_table database. The algorithm for "semact" is given below.

```
semact(in:Tnode,in:Num,in:Pnode)
  case Tnode sa_type of
    getid : idget(in:Idtype,in:Num,in:Pnode);
    choose: choice_node(in:Choicetype,in:Num,in:Pnode);
    more  : moresa(in:Moretype,in:Num,in:Pnode);
  end case
end semact.
```

(1) Semantic Action Idget

```
idget(in:Idtype,in:Num,in:Pnode)
  chk_symtab(in:Idtype,out:Val),
  assert "Val" as a terminal program node
    with "Num" as its index and "Pnode" as parent link,
  calculate "ColDiff",
  modify_righttreeRC(in:Pnode,in:0,in:ColDiff),
  chk_getid(in:Idtype,in:Val)
end idget.
```

Idget first calls "chk_symtab" to get a value for "Idtype" either from the symbol table (if it exists) or from the user. The syntax and the uniqueness of user supplied values for

identifiers are checked by this predicate. The user-obtained identifier value is asserted as the symbol table fact in the database. The difference in the column co-ordinate due to the addition of an identifier value is propagated. Additionally, all other program nodes requiring a value for the same identifier type are automatically expanded by "chk_getid".

(2) Semantic Action Choose

```
choice_node(in:Choicetype,in:Num,in:Pnode)
  get list of choices from choice fact
    c(in:Choicetype,out:Choicelist),

  menu(in:Topleftrow,in:TopleftCol,in:Choicelist,
                                   out:Choice),
  get_choice_nt(in:choice,in:1,in:Choicelist,out:Ntname),
  assert Ntname as a program fact
end choice_node.
```

The list of alternate grammar rules (left-side of grammar rules) is obtained by the choice fact in the database. The "Choice" is obtained as a position in the list from the user by the predicate "menu". The "menu" sets up a self adjusting pop-up window in which the size of the window is adjusted to display all the items in the "Choicelist". The pop-up window is positioned close to the program node being expanded. User chooses the "Choice" from the menu list by pointing to it and then hitting return. From the user "Choice" which is a position in the "Choicelist", the predicate "get_choice_nt" obtains the string value "Ntname" corresponding to that position. The chosen "Ntname" is asserted into the Prolog database as a program fact.

(3) Semantic Action More

```
moresa(in:Moretype,in:Num,in:Pnode)
  get left Not-terminal node "Lnode" of "Pnode",
  change "Lnode" index to "Num",
  assert program nodes "Moretype" of non_terminal type
    and format "nl" node between "Lnode" and "Pnode",
  adjust the co-ordinates of "Pnode" and all non-terminal
    nodes in the right_tree of "Pnode",
end moresa.
```

The execution of "moresa" results in the insertion of a non-terminal node, "Moretype", to the left of "Pnode". As an example, the execution of "moresa(in: TableProcs, in: Num, in: MoreTableProcs)" inserts a non-terminal node "<TableProcs>" to the left of "<MoreTableProcs>" node in the program tree.

(4) Next Node to be Expanded After Semantic Action

```
chk_more_choose(in:Tnode,in:Pnode,out:NewPnode)
  case Tnode_type of
    getid :expand_next(in:Pnode,out:NextNode),
           chk_expand(in:Pnode,in:NextNode,out:NewPnode);
    choose:expand_next(in:Pnode,out:NextNode),
           expand(in:NextNode,out:NewPnode); /*indirect
                                              recursion*/

    more  :get left non-terminal node "Lnode" of "Pnode"
           expand(in:Lnode,out:NewPnode); /*indirect
                                            recursion*/
  end case
end chk_more_choose.
```

The next node to be expanded is obtained through depth first sequencing by the predicate "expand_next". In the case of "getid" semantic action, "NextNode" may be null. In such cases, chk_expand returns "NewPnode" which is the same as "Pnode". The "choose" semantic action undergoes one more expansion of the

previously chosen non-terminal node before "NewPnode" is returned. For the case of the semantic action, "more", the previously-inserted non-terminal node ("Lnode") to the left of "Pnode" gets expanded once more before "NewPnode" is returned.

(5) Expanding Non-Terminal Node. (Other than those which derive Semantic Action Node)

The right side terms corresponding to the non-terminal node are obtained from the stored grammar rules and grafted into the program tree using "link_rhs(in:Tnode,in:Num,in:Pnode)". The change in screen co-ordinates is propagated to the right of "Pnode". The next node to be expanded, "NewPnode", is obtained through the predicates "expand_next" and "chk_expand".

4.4.6 Save_Program

The predicate "Save_pgm" first prompts the user for the disk file name. "Save_pgm" does a pre-order traversal of the program tree and writes all terminal nodes into the disk file. The format statement advances to the next line and indents the program text.

4.4.7 Program Tree

Figure 4.3 shows an example program tree stored as facts in the Prolog database. The various terms in the program facts are described in section 4.3.

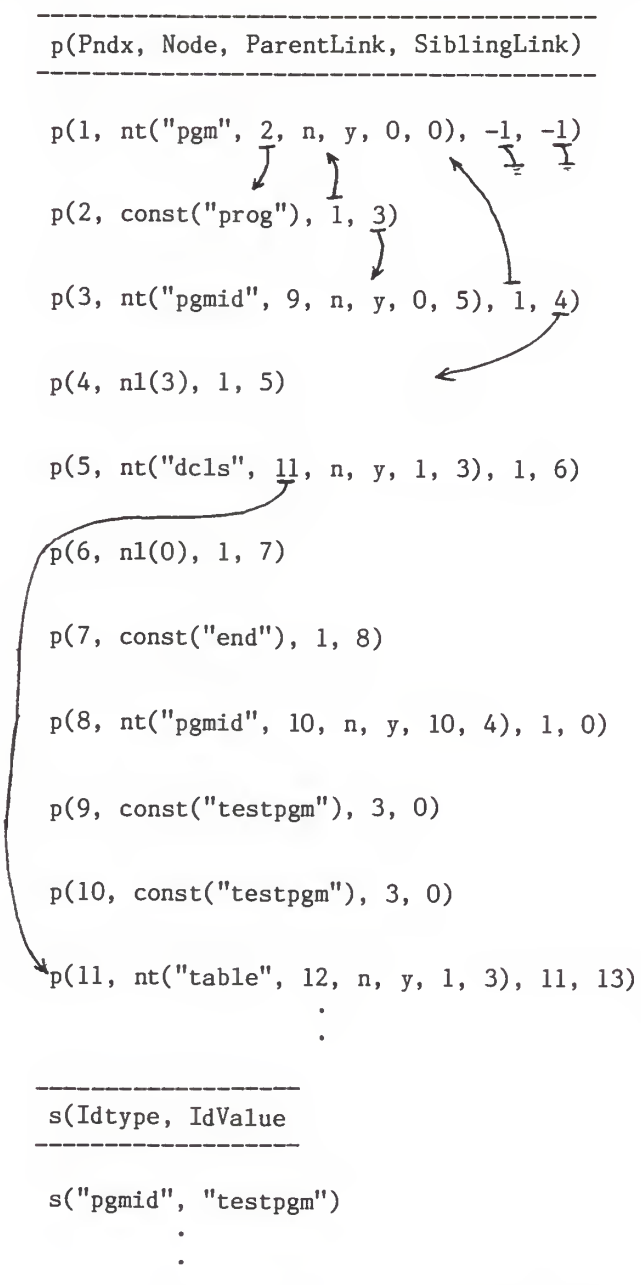


Figure 4.3. Dynamically asserted program tree

4.4.8 Program Display

The display of the program starts with the start symbol non-terminal being displayed within angle brackets and highlighted by an inverse video. The cursor is represented as an inverse video bar on the screen. The program generated is represented as a series of partially developed program displays on the screen.

The "display" is a utility predicate which is called whenever the underlying program structure is updated/changed or when the current program node to be displayed lies outside the screen window. The algorithm for "disp" is presented below.

```
disp(in:Pnode)
  clearwindow,
  assert "Pnode" as "CurrentNode",
  TempNode=1,
  calc_diff(out:diff),
  case Diff of
    0    :assert CursorRC as 0,0,
          set_disp_flag(in:true),
          StartNode=1,
          display_now(in:TempNode,in:StartNode);
    not 0:find_start(in:Pnode,out:StartNode,out:Indent),
          assert CursorRc as 0,indent ,
          set_display_flag(in:false),
          display_now(in:TempNode,In:StartNode);
  end case
end disp.
```

The predicate "calc_diff(out:Diff)" determines whether "Pnode" can be displayed from the beginning of the program tree. If "Diff" is equal to zero, "display_now" is invoked to display the program starting from the root node. If the current "Pnode" cannot be displayed starting from root node then "find_start"

gets the start program node from which the "display_flag" should be turned on (and so also the screen display).

Display_now performs preorder traversal always from the root and displays the nodes starting from the "StartNode" until the cursor co-ordinates exceed the screen window co-ordinates. "Pnode" is displayed at the near center of the window and is highlighted. The predicate "chk_write(in:Node)" writes the terms on the screen and highlights "Pnode" if it is "CurrentNode". Angle brackets are added while writing non-terminals whose "Visible" flag is on. The depth-first traversal ignores the children nodes of a non-terminal whose display flag is "on" although the node is expanded. This is the effect of the "ellipsis" command which hides the subtree display of a non-terminal program node.

4.5 Limitations and Extensions

The system implementation was intended only to demonstrate the feasibility of developing an interactive program generating system in a microcomputer environment using the Prolog language. The following text lists the system limitations and guidelines for future extensions.

- The programs are generated in a toy language. The target language should be a practical high level language suitable for library modules such as Modula or Ada.
- The grammar converter which converts input BNF style grammar representing program template into internal Prolog facts was not implemented. A detailed algorithm presented in this chapter for grammar converter should be implemented so that many more algorithmic program templates could be developed and interpreted to obtain tailor made program from the template.

- Additional module to convert a running module into template form in terms of grammar rules should be developed to avoid template coding in terms of grammar rules by programmers.

- Scope checking of identifiers is omitted in this implementation. A method of encoding scope information in the input grammar and a suitable interpretation should be implemented as an extension to the present work.

- Order of expansion of program nodes is left to users' choice. In some cases, the expansion of a node requires prior expansion of other nodes. Such semantic connections should be incorporated in the interpreter.

- Command "undo" for reverting back to the program structure and displaying the previous stage(s) should be implemented.

- The screen is redrawn for every modification of the program structure. A "smarter" display procedure should be implemented to enhance the system response.

4.6 Conclusions

Prolog offers a modular system development environment for problems involving the manipulation of grammar rules. The transformational approach appears to be the most productive method for automatic programming. Representation of algorithms for commonly used data structures in terms of grammar rules and refining the grammar rules to the specification at hand is a restrictive application of the transformational approach. Storage space problems and flexibility of the program generation process (interactive features) are the two major limitations in Prolog language systems. The present implementation effectively addresses these two issues through suitable internal representation in terms of database facts and interactive manipulation of the internal representation.

Ideally, a program generator should be a part of a software development environment within which many more functionalities and assistance features are provided to programmers in order to increase their productivity.

BIBLIOGRAPHY

[Bah 85] Bahlke, Rolf, Snelting, Gregor, "The PSG - Programming System Generator," ACM SIGPLAN Notices, pp 28-33, Vol. 28, No. 5, 1985.

[Bal 78] Balzer, Robert M., Goldman, Neil, Weile, David, "Informality in Program Specifications," IEEE Transations On Software Engineering, SE-4(2),1978, pp 94-10

[Bal 83] Balzer, Roberts, Cheatham, T.E., Green, Cordell, "Software Technology in 1990's: Using a New Paradigm," IEEE Computer, pp 39-45, Nov. 1983.

[Bar 79] Barstow, David R., "An Experiment in Knowledge-Based Automatic Programming," Artificial Intelligence 12, 1979: pp 73-119

[Bar 85] Barstow, David R., "On Convergence Toward a Database of Program Transformation," ACM Transactions on Programming Languages and Systems, Vol.7, No.1, Jan. 1985, pp 1-9.

[Barr 82] Barr, Aron and Feigenbaum, Edward A., Eds., The Handbookd of Artificial Intelligence, Vo.2, pp 297-375, William Kaufmann, Inc., Los Altos, CA.

[Bart 85] Barret, Kirk., "A Program Development System Using an Attribute Grammar," Master Report, Kansas State University, Manhattan, KS, 1985.

[Bau 79] Bauer, Michael A.,"Progrmming by Examples," Artificial Intelligence 12, 1979, pp 1-21

[Bie 84] Biermann, Allan W., Natural Language Programming, in Computer Program Synthesis Methodologies, Reidal publishing co., Dordrecht, Holland, eds. Biermann & Guiho, 1984, pp 335-368

[Bob 85] Bobrow, Daniel G. and Stefik, Mark J., "Perspectives on Artificial Intelligence Programming," Science 231, 1986, pp 951-957

[Bob 86] Bobrow, Daniel G., "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms," IEEE Transactions On Software Engineering, Vol. SE-11,No.11, Nov. 1985, pp 1401-1408.

[Boy 84] Boyle, J and Muralidh'aran, M., "Program Reusability Through Program Transformation," IEEE Transactions of Software Engineering, Vol.10, No.5, Sept. 1984: pp 574-588

[Che 84] Cheatham, Thomas E., "Reusability Through Program Transformation," IEEE Transactions on Software Engineering, SE-10(5), 1984: pp 589-594

[Clo 84] Clocksin,W.F, and Mellish C.S., Programming in Prolog. Springer-Verlag, New York, 1984.

[Der 85] Dershowitz, N, "Program Abstraction and Instantiation," ACM Transactions on Programming Languages and Systems, Vol 7, #3, pp 446-477, Jul. 1985

[Fea 82] Feather, Marin S. and London, Philip E., "Implementing Specification Freedoms," Science of Computer Programming 2, pp 9-31, Amsterdam: North-Holland Publishing Company, 1982.

[Fre 85] Frenkel, Karen A., "Toward Automating the Software Development Cycle," Communications of ACM, pp 578-589, Vol.28, No.6, Jun. 1985

[Gre 77] Green, C., "A Summary of PSI Program Synthesis System," IJCAI 5, PP 380-381, 1977

[Kan 81] Kant, Elaine and Barstow, David R., "The Refinement Paradigm: The Interaction of Coding and Efficiency of Knowledge in Program Synthesis," IEEE Transaction of Software Engineering, Vol.SE-7, No.5, Sept.1981: pp 458-471

[Man 78] Manna, Z, and Waldinger, R., DEDALUS - The DEDuctive ALgorithm Ur-Synthesizer, National Computer Conference, Anaheim, CA, 1978: pp 683-690

[Man 80] Manna, Zohar and Waldinger, Richard, "A Deductive Approach to Program Synthesis," ACM Transactions on Programming Languages and Systems, 2(1), pp 90-121, 1980

[Mun 86] Munakata, Toshinori, "Procedurally Oriented Programming Techniques in Prolog," IEEE EXPERT, Summer 1986, pp 41-47.

[Ols 86] Olsen, Dan R. Jr., "Editing Templates: A User Interface Generation Tool," IEEE Computers Graphics and Applications, pp 40-46, Nov. 1986.

[Par 83] Partch, H. and Steinbruggen, R., "Program Transformation System," ACM Computing Surveys, Vol 15, #3, 1983:pp 199-236

[Pea 86] Peak, Marita E., "A Prolog Prototype of A Module Development System," Masters Thesis, Kansas State University, Manhattan, KS, 1986.

[Phi 77] Phillip, J.V., "Program Inference from Traces using Multiple Knowledge Sources," IJCAI, 812, 1977

[Rei 85] Reiss, Steven P., "PECAN: Program Development Systems that Support Multiple Views," IEEE Transactions on Software Engineering, pp 276-285, Vol.SE-11, No.3, Mar. 1985.

[Rep 84] Reps, Thomas, Teitelbaum, Tim, "The Synthesizer Generator," ACM SIGPLAN Notices, pp 28-33, Vol.19, No.5, 1984.

[Rut 78] Ruth, G., "Protosystem I, an Automatic Program System Prototype," Proceedings of the National Computer Conference, Anaheim, CA AFIPS 47: pp 675-681, 1981

[Shn 83] Shneiderman, Ben, "Direct Manipulation: A Step Beyond Programming Languages," IEEE Computer, Aug. 1983, pp 57-69.

[Smi 85] Smith, Douglas R., Kotic, Gordon V., and Westfold, Stephan J., "Research on Knowledge-Based Software Environment at Kestrel-Institute," IEEE Transactions on Software Engineering, Vol.SE-11, No.11, Nov.1985: pp 1278-1295

[Ste 86] Sterling, Leon and Shapiro, Ehud. The Art of Prolog, Advanced Programming Techniques. The MIT press, London. 1986.

[Sub 85] Subramanyam, P.A., "The Software Engineering of Expert Systems: Is Prolog Appropriate?," IEEE Transactions on Software Engineering, Vol.Se-11, No.11, Nov.1986, pp 1391-1400.

[Sum 77] Summers, Phillip D., "A Methodology for Lisp Program Construction from Examples," JACM 24(1), 1977, pp 161-175

[Tei 81] Teitelbaum, Tim, Reps, Thomas, Horwitz, Susan, "The Why and Wherefore of the Cornell Program Synthesizer," ACM SIGPLAN, pp 8-16, Vol.16, No.6, Jun. 1981.

[Tur 86] Turbo-Prolog user manual, Borland International Inc., Scotts valley, CA, 1986, 221 pages.

[War 80] Warren, David H.D., "Logic Programming and Compiler Writing," Software-Practices and Experience, Vol.10, pp 97-125, 1980.

[Wat 82] Waters, Richard C., "The Programmer's Apprentice: Knowledge Based Program Editing," IEEE Transactions on Software Engineering, Vol.SE-5., No.1, Jan. 1982.

[Wat 85] Waters, Richard C., "The Programmer's Apprentice: A session with KBEmacs," IEEE Transactions On Software Engineering, SE-11(11), 1985, pp 1296-1320.

[Won 86] Wong, William G., "Prolog: A Language for Artificial Intelligence," PC Magazine, Oct.14, 1986, pp 247-263.

****

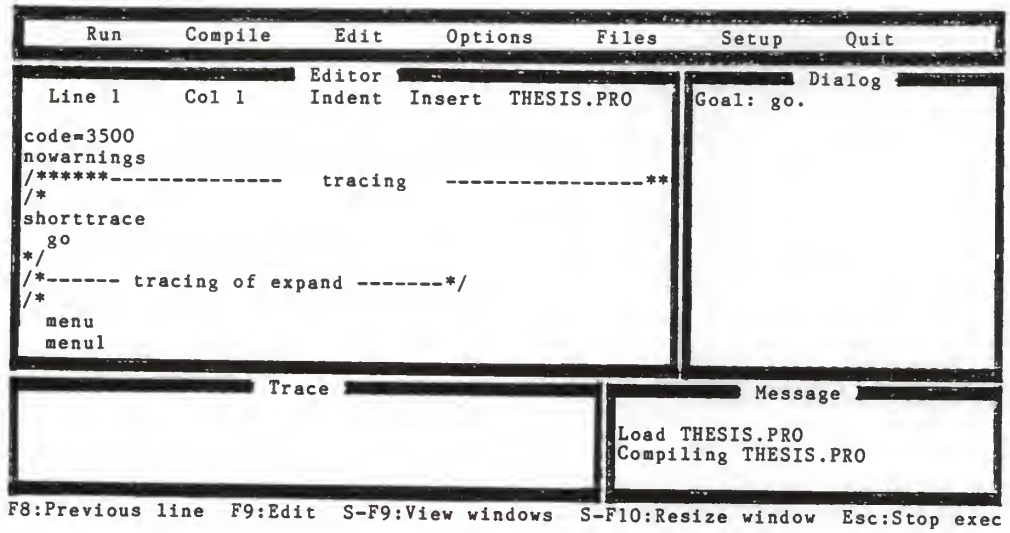
Additional reference

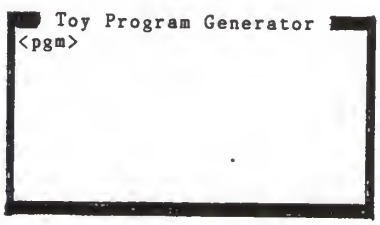
[Ric 86] Rich, Charles, Waters, Richard C., Editors, Readings in Artificial Intelligence and Software Engineering, Morgan Kaufman Publishers, Inc., CA 94022.

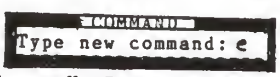
Appendix I

SAMPLE TERMINAL SESSION

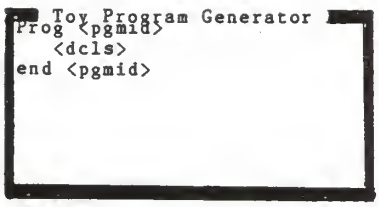
The following pages present a hard-copy listing of an actual terminal session using Prototype Program Generator with the sample grammar. The current "node" is actually highlighted with an inverse video bar, but this highlight is not transimitted to the paper while printing.



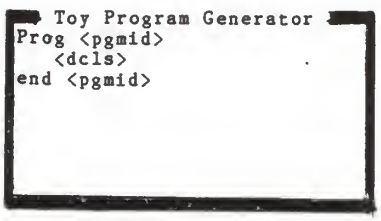


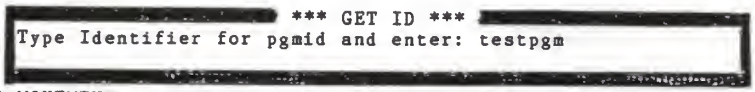


STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save

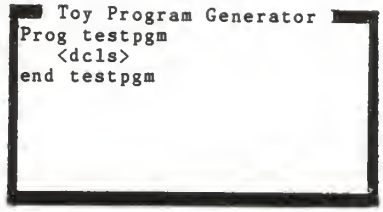


STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save

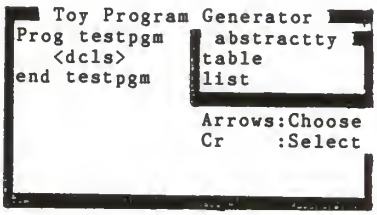




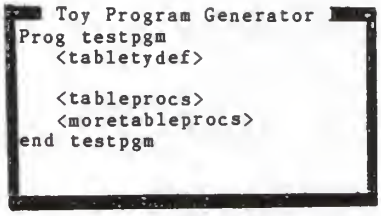
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save



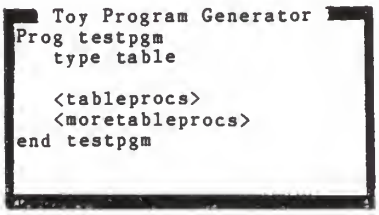
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save



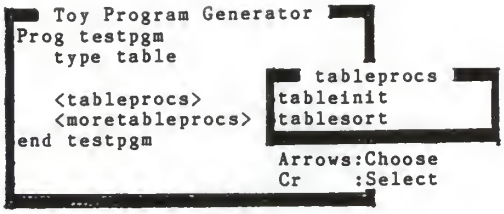
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save



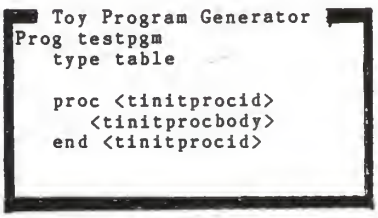

```
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis  'q':Quit 's':Save
```



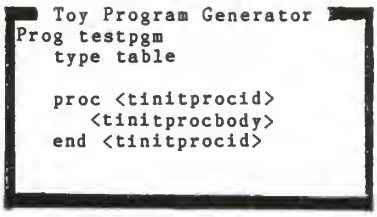

```
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis  'q':Quit 's':Save
```

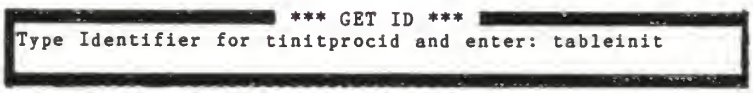


STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save

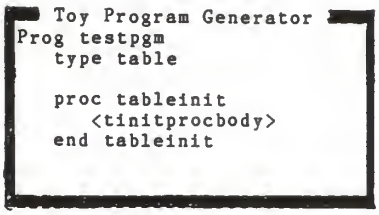


STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save

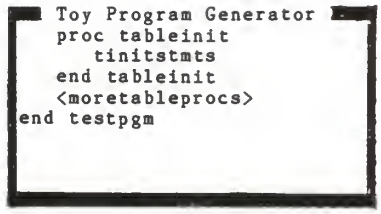




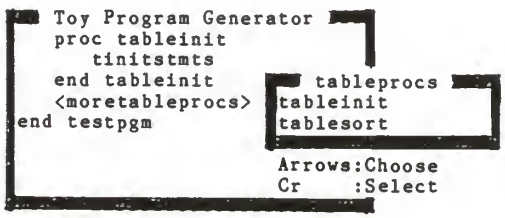

```
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis  'q':Quit 's':Save
```



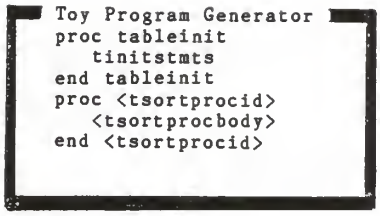

```
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis  'q':Quit 's':Save
```



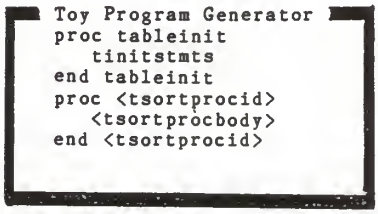
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save

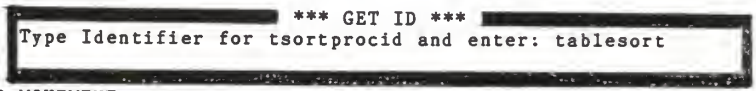


STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save

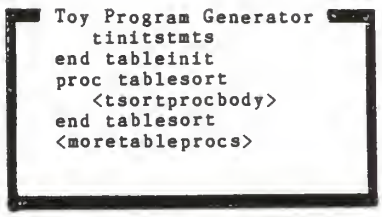


STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save

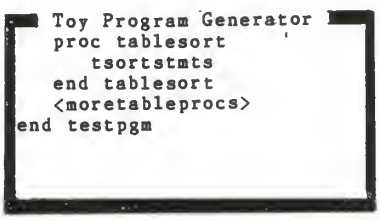




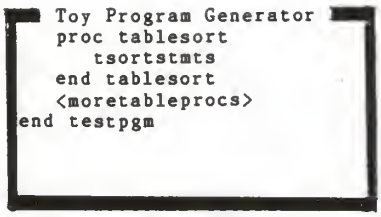
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand '.':Ellipsis 'o':Open-Ellipsis 'q':Quit 's':Save




```
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis  'q':Quit 's':Save
```




```
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis  'q':Quit 's':Save
```




```
STRUCTURED MOVEMENT - ARROWS ==> Up:Parent Down:Child Left,Right:Sibling Nt's
COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis  'q':Quit 's':Save
```

FINAL PROGRAM

```
Prog testpgm
   type table

   proc tableinit
      tinitstmts
   end tableinit
   proc tablesort
      tsortstmts
   end tablesort

end testpgm
```

Appendix II

TEST GRAMMAR

```
<pgm> --> 'Prog' <pgmid> nl(3) <dcls> nl(0) 'end' <pgmid>/

<pgmid> --> .id(pgmid)./

<dcls> --> .choose(abstractty, [table, list])./

/****** TABLE *****/

<table> --> <tabletydef> nl(0) nl(3) <tableprocs> nl(3)
            <moretableprocs>/

<tabletydef> --> 'type table'/

<tableprocs> --> .choose(tableprocs [tableinit, tablesort])./

<moretableprocs> --> .more(tableprocs)./

/****** TABLE INIT *******/

<tableinit> --> 'proc' <tinitprocid>  nl(6) <tinitprocbody>
                nl(3) 'end' <tinitprocid>/

<tinitprocid> --> .id(tinitprocid)./

<tinitprocbody> --> 'tinitstmts'/

/**** TABLE SORT *****/

<tablesort> --> 'proc' <tsortprocid>  nl(6) <tsortprocbody>
                nl(3) 'end' <tsortprocid>/

<tsortprocid> --> .id(tsortprocid)./

<tsortprocbody> --> 'tsortstmts'/
```

```
/****** LIST *****/

<list> --> <listtydef> nl(0) nl(3) <listprocs> nl(3)
             <morelistprocs>/

<listtydef> --> 'type list'/

<listprocs> --> .choose(listprocs [listinit, listinsert])./

<morelistprocs> --> .more(listprocs)./


/****** LIST INIT *******/

<listinit> --> 'proc' <linitprocid>  nl(6) <linitprocbody>
               nl(3) 'end' <linitprocid>/

<linitprocid> --> .id(linitprocid)./

<linitprocbody> --> 'linitstmts'/


/**** LIST INSERT *****/

<listinsert> --> 'proc' <linsertprocid>  nl(6) <linsertprocbody>
                 nl(3) 'end' <linsertprocid>/

<linsertprocid> --> .id(linsertprocid)./

<linsertprocbody> --> 'linsertstmts'/
```

Appendix III

PROGRAM LISTING

```
code=3500
nowarnings

/*--------------- domains for readkey -------------------*/
domains
  file = pgmfile
  key = cr;esc;break;tab;btab;del;bdel;ins;end;home;
        fkey(integer);up;down;left;right;char(CHAR);other

/*--------------- domains for display -------------------*/
domains
   dbvars = progndx; curr_RC; dbrowcol; tempRC; diff;
            maxline; p; curr_str; disp_flag; curr_num;
            s
  visible,expanded = y;n /* for nt's */
  t_or_f = t;f /* for display flag */
  node = nt(symbol,integer,visible,expanded,integer,integer);
           /* nt(ntname,chldndx,vis,exp,plink,slink) */
         const(string);
         sa(saname,symbol);
           /* sa(saname,satype) */
         nl(integer)
           /* nl(abstab) */

  dbfact =  progndx(integer);
            maxline(integer);
            curr_RC(integer,integer);
            dbrowcol(integer,integer);
            tempRC(integer,integer);
            diff(integer);
            p(integer,node,integer,integer);
            curr_str(string);
            disp_flag(t_or_f);
            d(symbol,integer);
            t(integer,tnode,integer);
            c(symbol,symlist);
            curr_num(integer);
            s(symbol,string)

/*--------------- domains for menu system ---------------*/
domains
  symlist = symbol*
```

```
/*---------------- domains for expansion ----------------*/
domains
saname = getid;
         retrid;
         choose;
         more
tnode = nt(symbol);
        const(string);
        nl(integer);
        sa(saname,symbol)

/*------------ database for display routines -------------*/
database
  progndx(integer)
      /* current prog_index under development */
      /* Progndx(Pndx) */

  maxline(integer)
      /* number of lines for one screen */
      /* maxline(line) */

  curr_RC(integer,integer)
      /* curr_RC(row,col) */
      /* row col of current node */

  dbrowcol(integer,integer)
      /* dbrowcol(row,col) */
      /* row col of cursor on screen */

  tempRC(integer,integer)
      /* tempNdxRC(pndx,row,col) */
      /* row,col of temp. prog index.
         used in finding row col of a given
         index, or given row, find the first
         prog index which has that row coordinate */

  diff(integer)
      /* diff of data structure Row and
         current screen row              */

  p(integer,node,integer,integer)
      /* program node */
      /* p(p_node_ndx,node,parent_link,sibling_link) */

  curr_str(string)
      /* used to highlight curr_str
         while structural movement */

  disp_flag(t_or_f)
      /* used to start displaying in display_now */
```

```
/*--------------- database for expand -----------------*/
  d(symbol,integer)
    /* d(ntname,rhsndx) */
    /* definition of nt */

  t(integer,tnode,integer)
    /* t(index,tnode,siblingndx) */
    /* tokens in the database */

  c(symbol,symlist)
    /* c(choicetype,choicelist) */
    /* chice list for sa choose */

  curr_num(integer)
    /* current max prog index */

  s(symbol,string)
    /* s(Idtype,IdVal) */

/*-------------------------------------------------------*/
/*                  --- music ---                        */
/*-------------------------------------------------------*/
domains
    pitch=high;low

predicates
      music(pitch,integer)
      bell

clauses
bell:-
  music(high,500).

music(high,F):-
   F<1500,!,sound(1,F),
   F1=F+300,music(high,F1).
music(high,F):-
  music(low,F).

music(low,F):-
  F>500,!,/* sound(1,F), */
  F1=F-300,music(low,F1).
music(low,_).


/*-------------------------------------------------------*/
/*                    --- readkey ---                    */
/*-------------------------------------------------------*/
predicates
  readkey(key)
     /* readkey(out:key) */
```

```
  key_code(key,char,integer)
     /* key_code(out:Key, in:asciikey, in:key_int) */

  key_code2(key,integer)
     /* key_code2(out:key, in:key_int) */

clauses
  readkey(Key):-
    readchar(T),char_int(T,Val),key_code(Key,T,Val).

  key_code(Key,_,0):-
    readchar(T),char_int(T,Val),key_code2(Key,Val),!.

  key_code(break,_,3):-!.
  key_code(bdel,_,8):-!.
  key_code(tab,_,10):-!.
  key_code(cr,_,13):-!.
  key_code(esc,_,27):-!.
  key_code(char(T),T,_):-!.

  key_code2(btab,15):-!.
  key_code2(home,71):-!.
  key_code2(up,72):-!.
  key_code2(left,75):-!.
  key_code2(right,77):-!.
  key_code2(end,79):-!.
  key_code2(down,80):-!.
  key_code2(ins,82):-!.
  key_code2(del,83):-!.
  key_code2(fkey(N),V):- V>58, V<70, N=V-58,!.
  key_code2(other,_).

/*-------------------------------------------------------*/
/*                    --- utils ---                      */
/*-------------------------------------------------------*/
predicates
  str_symbol(string,symbol)
clauses
  str_symbol(X,Y):-X=Y.

/*-------------------------------------------------------*/
/*                    --- display ---                    */
/*-------------------------------------------------------*/
/*---------- Predicates for display routines ----------*/
predicates
  get_dbvar(dbfact)
      /* gets the first value  */

  retract_fst(dbfact)
      /* retracts first occurance of db fact */
```

```
calc_diff(integer)
    /* calc_diff(out:Diff) */

brace_nt(symbol,string)
    /* brace_nt(in:ntname, out:braced_ntstr) */
    /* puts angle brackets to nt */

del_all(dbvars)
    /* del_all(in: dbvarname) */
    /* retracts all asserted facts of a
       db global var predicate          */

set_progndx(integer)
    /* update_progndx(in:Pndx) */

update_currStr(string)
    /* update_currStr(in:Str) */

disp(integer)
    /* disp(in: curr_node_ndx) */
    /* displays the program nodes including
       p(index,_,_,_). If the node of interest
       is within screen RC, display from the
       begining. Else find start index,
       difference starting from start index.
       Diff will be used to calculate
       screen pos of nt's & constants */

getRC(integer,integer,integer)
    /* get row & col of p(index,..) */
    /* getRC(in:curr_ndx, out:row, out:col) */

getlRC(integer,integer,integer)
    /* get row & col similar to getRC except
       p node nt case being tested. Used in
       expand predicate                    */
    /* getlRC(in:curr_ndx, out:row, out:col) */

find_start(integer,integer,integer)
    /* find_start(in:Cndx,out:Sndx,out:Indent) */
    /* find Sndx such that Row of Sndx-Diff=0 */

get_fst_parent(integer,integer)
    /* get_fst_parent(in:Cndx,out:Parndx) */
    /* Parndx is first parent which falls outside
       current screen. i.e. Row(Parndx)-Diff<0  */
```

```
  computeRC(integer,integer,integer,integer)
      /* computeRC(in:tempndx, in:curr_ndx,
                   out:currR, out:currC) */
      /* when second index = first index then
         second row & col will get the value
         from first row & col from db tempRC. */

  getlen(integer,integer)
      /* getlen(in:index, out:length) */
      /* length of nt or const. nl(tab) returns -ve length */

  display_now(integer,integer)
      /* display_now(in:Tempndx,in:Start_ndx) */
      /* display from start index.  Highlight
         current index node.  diff helps in
         computing the screen position from
         the stored RC of p nodes. */

  updatedbrc(integer,integer)
      /* updatedbrc(inout:row, inout: col) */

  setdbcol(integer)
      /* setdbcol(in:col) */

  setdbrow(integer)
      /* setrow(in:row) */

  chk_write(string)
      /* chk_write(in:string) */
      /* check the currRC=dbRC.
       if so, highlight while writing
       use diff to adjust currRC */

  set_disp_flag(t_or_f)
      /* set_disp_flag(in:TF) */

  update_disp_flag(integer,integer)
      /* update_disp_flag(in:Tempndx,in:Startndx) */
      /* sets dbase disp_flag(t) if Tempndx=Startndx */
  next_line
      /* update dbrowcol by a row and change
         cursor position                    */

/*---------- clauses for display routines ----------*/
clauses

maxline(6).

/***** del_all *****/
del_all(disp_flag):-
  retract(disp_flag(_)),fail.
del_all(disp_flag):- !.
```

```
del_all(dbrowcol):-
  retract(dbrowcol(_,_)),fail.
del_all(dbrowcol):- !.

del_all(curr_RC):-
  retract(curr_RC(_,_)),fail.
del_all(curr_RC):- !.

del_all(curr_str):-
  retract(curr_str(_)),fail.
del_all(curr_str):- !.

del_all(tempRC):-
  retract(tempRC(_,_)),fail.
del_all(tempRC):- !.

del_all(diff):-
  retract(diff(_)),fail.
del_all(diff):- !.

del_all(maxline):-
  retract(maxline(_)),fail.
del_all(maxline):- !.

del_all(progndx):-
  retract(progndx(_)),fail.
del_all(progndx):- !.

del_all(p):-
  retract(p(_,_,_,_)),fail.
del_all(p):- !.

del_all(curr_num):-
  retract(curr_num(_)),fail.
del_all(curr_num):- !.

del_all(s):-
  retract(s(_,_)),fail.
del_all(s):- !.

/***** get_dbvar *****/
/** gets the first value of global db fact */
get_dbvar(p(Ndx,Node,P1,S1)):-p(Ndx,Node,P1,S1),!.
get_dbvar(d(Ndx,R1)):-d(Ndx,R1),!.
get_dbvar(t(Ndx,Tnode,S1)):-t(Ndx,Tnode,S1),!.
get_dbvar(dbrowcol(Row,Col)):-dbrowcol(Row,Col),!.
get_dbvar(curr_RC(Row,Col)):-curr_RC(Row,Col),!.
get_dbvar(tempRC(Row,Col)):-tempRC(Row,Col),!.
get_dbvar(disp_flag(TrueFalse)):-disp_flag(TrueFalse),!.
get_dbvar(diff(Diff)):-diff(Diff),!.
get_dbvar(curr_str(Str)):-curr_str(Str),!.
```

```
get_dbvar(progndx(Ndx)):-progndx(Ndx),!.
get_dbvar(maxline(Ndx)):-maxline(Ndx),!.

/***** retract_fst *****/
retract_fst(p(Ndx,Node,P1,S1)):-
  retract(p(Ndx,Node,P1,S1)),!.

/***** brace_nt ******/
brace_nt(Nt1,Nt2):-
  concat("<",Nt1,N),concat(N,">",Nt2).

/******* update_currStr ********/
update_currStr(Str):-
  del_all(curr_str),asserta(curr_str(Str)).

/******* set_progndx ********/
set_progndx(Pndx):-
  del_all(progndx),asserta(progndx(Pndx)).

/***** chk_write *****/
chk_write(Str):-
  get_dbvar(diff(Diff)),
  get_dbvar(dbrowcol(R,C)),get_dbvar(curr_RC(R1,C1)),
  R2=R1-Diff,R=R2,C=C1,!,
  Str_len(Str,Len),
  update_currStr(Str),
  field_attr(R,C,Len,28),
  field_str(R,C,Len,Str).
chk_write(Str):-
  write(Str).

/****** calc_diff ******/
calc_diff(Diff):-
  get_dbvar(curr_RC(R,_)),get_dbvar(maxline(L)),R<L,!,
  Diff=0,del_all(diff),asserta(diff(Diff)).

calc_diff(Diff):-
  get_dbvar(curr_RC(R,_)),get_dbvar(maxline(L)),
  Diff=R-(L-3),del_all(diff),asserta(diff(Diff)).

/****** next_line ******/
next_line:-
  !,setdbcol(0),updatedbrc(1,0),
  get_dbvar(dbrowcol(Rf,Cf)),cursor(Rf,Cf).

/****** set_disp_flag ******/
set_disp_flag(TF):-
  del_all(disp_flag),asserta(disp_flag(TF)).
```

```
/****** update_disp_flag ******/
update_disp_flag(Tempndx,Startndx):-
  Tempndx=Startndx,!,set_disp_flag(t).
update_disp_flag(_,_).

/****** disp *****/
disp(Curr_ndx):- /* p(Curr_ndx,....) should be
                      either nt or const        */

  shiftwindow(1),clearwindow,
  getRC(Curr_ndx,R,C), /* get row col of current node */
  del_all(curr_RC), /* deletes all curr_RC if present */
  asserta(curr_RC(R,C)),
  del_all(dbrowcol), /* deletes all dbrowcol if present */
  asserta(dbrowcol(0,0)), /*display starts from top left */
  calc_diff(Diff),
  Diff=0,!,set_disp_flag(t),
  Start_ndx=1, /* display from begining */
  Tempndx=1,display_now(Tempndx,Start_ndx),next_line,!.

disp(Curr_ndx):-
  find_start(Curr_ndx,Start_ndx,Indent),
                              /* get window start index */
  del_all(dbrowcol),
  asserta(dbrowcol(0,Indent)), /* maintain the indentation of
                                  start token */
  set_disp_flag(f),Tempndx=1,
  display_now(Tempndx,Start_ndx),next_line,!.

/***** find_start *****/
find_start(Cndx,Sndx,Indent):-
  get_fst_parent(Cndx,Parndx),
  get_dbvar(p(Parndx,nt(_,C1,_,_,Rp,Cp),_,_)),
  del_all(tempRC),asserta(tempRC(Rp,Cp)),
  get_dbvar(diff(Diff)),computeRC(C1,Sndx,Diff,Indent),!.

/***** get_fst_parent *****/
get_fst_parent(Cndx,Parndx):-
  get_dbvar(p(Cndx,_,Parndx,_)),
  get_dbvar(p(Parndx,nt(_,_,_,_,Rp,_),_,_)),
  get_dbvar(diff(Diff)),(Rp-Diff)<0,!.
get_fst_parent(Cndx,Parndx):-
  get_dbvar(p(Cndx,_,C1,_)),!,
  get_fst_parent(C1,Parndx).

/***** getRC *****/
getRC(Curr_ndx,R,C):-
  get_dbvar(p(Curr_ndx,nt(_,_,_,_,R,C),_,_)),!.

getRC(Curr_ndx,R,C):-getlRC(Curr_ndx,R,C).
```

```
/***** getlRC *****/
getlRC(Curr_ndx,R,C):-
  get_dbvar(p(Curr_ndx,_,Pndx,_)),
  get_dbvar(p(Pndx,nt(_,Chldndx,_,_,PR,PC),_,_)),
  del_all(tempRC),asserta(tempRC(PR,PC)),
  computeRC(Chldndx,Curr_ndx,R,C),!.

/***** computeRC *****/
computeRC(_,Cndx,Cr,Cc):-
  bound(Cr),bound(Cndx),bound(Cc),!.

computeRC(Vndx,_,_,_):-
  Vndx<=0,!.  /* trying to follow null sibling link */

computeRC(Vndx,Cndx,Cr,Cc):-
  bound(Cr),get_dbvar(tempRC(Vr,Vc)),Vr=Cr,!,
  Cndx=Vndx,Cc=Vc.

computeRC(Vndx,Cndx,Cr,Cc):-
  bound(Cndx),Vndx=Cndx,!,
  get_dbvar(tempRC(Vr,Vc)),Cr=Vr,Cc=Vc.

computeRC(Vndx,Cndx,Cr,Cc):-
  get_dbvar(p(Vndx,nl(Tab),_,Nvndx)),!,
  get_dbvar(tempRC(Vr,_)),Vrl=Vr+1,Vcl=Tab,del_all(tempRC),
  asserta(tempRC(Vrl,Vcl)),
  computeRC(Nvndx,Cndx,Cr,Cc).

computeRC(Vndx,Cndx,Cr,Cc):-
  get_dbvar(p(Vndx,const(Str),_,Nvndx)),!,
  str_len(Str,Len),
  get_dbvar(tempRC(Vr,Vc)),Vcl=Vc+Len+1,del_all(tempRC),
  asserta(tempRC(Vr,Vcl)),
  computeRC(Nvndx,Cndx,Cr,Cc).

computeRC(Vndx,Cndx,Cr,Cc):-
  get_dbvar(p(Vndx,nt(Str,_,y,_,_,_),_,Nvndx)),!,
  str_len(Str,Len),get_dbvar(tempRC(Vr,Vc)),
  del_all(tempRC),
  Vcl=Vc+Len+3, /* 2+1: 2 for braces, 1 for blank */
  asserta(tempRC(Vr,Vcl)),
  computeRC(Nvndx,Cndx,Cr,Cc).

computeRC(Vndx,Cndx,Cr,Cc):-
  get_dbvar(p(Vndx,nt(_,Cl,n,_,_,_),_,Nvndx)),!,
  computeRC(Cl,Cndx,Cr,Cc),
  computeRC(Nvndx,Cndx,Cr,Cc).
```

```
/***** getlen *****/
getlen(Ndx,Len):-
  get_dbvar(p(Ndx,nl(Tab),_,_)),!,Len=-Tab.
getlen(Ndx,Len):-
  get_dbvar(p(Ndx,nt(Ntname,_,_,_,_,_),_,_)),!,
  str_len(Ntname,Len1),Len=Len1+2.

getlen(Ndx,Len):-
  get_dbvar(p(Ndx,const(Const),_,_)),!,
  str_len(Const,Len1),Len=Len1.

/***** display_now *****/
/* display_now stops when cursor row=max window line */
display_now(_,_):-
  get_dbvar(maxline(X)),get_dbvar(dbrowcol(R,_)),
  R>=X,!.

display_now(Tempndx,_):-
  Tempndx<=0,!. /* trying to follow null link */

display_now(Tempndx,Start_ndx):-
  /* nt not visible,expanded */
  get_dbvar(p(Tempndx,nt(_,C1,n,y,_,_),_,S1)),!,
  update_disp_flag(Tempndx,Start_ndx),
  display_now(C1,Start_ndx),
  display_now(S1,Start_ndx).

display_now(Tempndx,Start_ndx):-
  /* either nt visible or const or nl but not yet
     reached start_ndx and hence not displayed */
  get_dbvar(p(Tempndx,_,_,S1)),
  update_disp_flag(Tempndx,Start_ndx),
  get_dbvar(disp_flag(f)),!,
  display_now(S1,Start_ndx).

display_now(Tempndx,Start_ndx):-
  /* nt visible */
  get_dbvar(p(Tempndx,nt(Ntname,_,y,_,R,C),_,S1)),
  !,get_dbvar(diff(Diff)),
  Newr=R-Diff,cursor(Newr,C),brace_nt(Ntname,Nt),
  chk_write(Nt),
  str_len(Ntname,Slen),Ntlen=Slen+3,
  updatedbrc(0,Ntlen),
  display_now(S1,Start_ndx).

display_now(Tempndx,Start_ndx):-
  get_dbvar(p(Tempndx,const(Constname),_,S1)),!,
  get_dbvar(dbrowcol(R,C)),cursor(R,C),
  chk_write(Constname),
  str_len(Constname,Slen),Constlen=Slen+1,
  updatedbrc(0,Constlen),
  display_now(S1,Start_ndx).
```

```
display_now(Tempndx,Start_ndx):-
  get_dbvar(p(Tempndx,nl(AbsTab),_,S1)),!,
  updatedbrc(1,0),setdbcol(AbsTab),
  display_now(S1,Start_ndx).

/***** updatedbrc *****/
updatedbrc(R,C):-
  retract(dbrowcol(R1,C1)),!,
  R2=R1+R,C2=C1+C,
  asserta(dbrowcol(R2,C2)).

/***** setdbcol *****/
setdbcol(C):-
  retract(dbrowcol(R1,_)),!,asserta(dbrowcol(R1,C)).

/***** setdbrow *****/
setdbrow(R):-
  retract(dbrowcol(_,C1)),!,asserta(dbrowcol(R,C1)).


/*-------------------------------------------------------*/
/*                  --- interpreter ---                  */
/*-------------------------------------------------------*/

/*------------- Predicates for interpreter --------------*/
predicates

  go

  interp
     /* interp(inout:pindex, inout:cmd) */

  chk_interp(key)
     /* chk_interp(in:key) */

  process_cmd(key,integer,integer)
     /* process_cmd(in:Cmd,in:Pndx,out:Npndx) */

  is_cursor(key)
     /* is_cursor(in:key) */

  is_ellipsis(key)
     /* is_ellipsis(in:key) */

  is_save(key)
     /* is_save(in:key) */

  is_out_ellipsis(key)
     /* is_out_ellipsis(in:key) */
```

```
update_currRC(integer,integer)
   /* update_currRC(in:Row,in:Col) */

chk_dummyPar(integer,integer)
   /* chk_dummyPar(in:Ndx,out:Parndx) */

hide_children(integer,integer)
   /* hide_children(in:Pndx,out:Newpndx) */
   /* ellipsis which doesn't display children */

open_children(integer,integer)
   /* open_children(in:Pndx,out:Newpndx) */
   /* expand ellipsis so as to display children */

struc_move(key,integer,integer)
   /* struc_move(in:Cursor,in:Pndx,out:Newpndx) */

move_out(integer,integer)
   /* move_out(in:Pndx,out:Newpndx) */

move_in(integer,integer)
   /* move_in(in:Pndx,out:Newpndx) */

move_right(integer,integer)
   /* move_right(in:Pndx,out:Newpndx) */

move_left(integer,integer)
   /* move_left(in:Pndx,out:Newpndx) */

get_rightnt(integer,integer)
   /* get_rightnt(in:Pndx,out:Newpndx) */

get_leftnt(integer,integer)
   /* get_leftnt(in:Pndx,out:Lpndx) */

chk_rightnt(integer,integer,integer)
   /* chk_rightnt(in:Chldndx,in:Pndx,out:Lpndx) */

get_fst_ntchld(integer,integer)
   /* get_fst_ntchld(in:Pndx,out:Cntndx)*/

get_last_ntchld(integer,integer)
   /* get_last_ntchld(in:Pndx,out:Lntchld) */

get_final_ntrc(integer,integer,integer)
   /* get_final_ntrc(in:Ntndx,out:Rf,out:Cf) */

restRC(integer,integer,integer,integer,integer)
   /* restRC(in:S,in:Rs,in:Cs,out:Rf,out:Cf) */

chk_display(integer)
   /* chk_display(in:Pndx) */
```

```
  chk_in_disp(integer,integer)
     /* chk_in_disp(in:Pndx,in:Newpndx) */

  get_fst_str(integer,string)
     /* get_fst_str(in:Pndx,out:Str) */

  update_curr_str(string)
     /* update_curr_str(in:Str) */

  modify_RC(integer,integer,integer)
     /* modify_RC(in:Pndx,in:Rd,in:Rc) */

  chk_uptree(integer,integer,integer)
     /* chk_uptree(in:Pndx,in:Rd,in:Cd) */

  modify_righttreeRC(integer,integer,integer)
     /* modify_righttreeRC(in:Pndx,in:Rd,in:Cd) */

  mod_col(integer,integer,integer,integer)
     /* mod_col(in:R,in:C,in:Cd,out:C1) */

  repeat
     /* provides new goal for failure driven loops */

  expand(integer,integer)
    /* expand(in:Ndx, out:Newndx)   */
    /* expands the program node Ndx if it is nt and
       not already expanded.  Newndx is prog index
       to be expanded next                          */

  save_pgm
    /* writes the program to a file */

  save_all(integer)
    /* save_all(in:Pndx) */

  chk_syntax(string,string)
    /* chk_syntax(in:Val1,out:Val2) */

  write_blanks(integer,integer)
    /* write_blanks(in:Tab,in:S) */


/*------------- clauses for interpreter --------------*/

clauses

/**** repeat *****/
repeat.
repeat:- repeat.
```

```
/**** go *****/
/** start the interpreter with help **/
go:-
  consult("toktree.dba"),
  del_all(curr_num),Ndx=1,
  assertz(curr_num(Ndx)),  /* used for unique program index */
  del_all(p),assertz(p(Ndx,nt("pgm",-1,y,n,0,0),-1,-1)),
  del_all(progndx),set_progndx(Ndx),
  del_all(s),              /* symbol table values */
  makewindow(99,7,0,"",0,0,25,80),
  makewindow(1,7,7," Toy Program Generator ",0,0,10,30),
  disp(Ndx),
  makewindow(21,112,0,"",23,0,2,80),

  write("STRUCTURED MOVEMENT - ARROWS ==>
         Up:Parent Down:Child Left,Right:Sibling Nt's"),
  nl,
  write("COMMANDS: 'e':Expand  '.':Ellipsis  'o':Open-Ellipsis
         'q':Quit 's':Save"),
  interp.

/**** interp ****/
/** quit interp if command is char('q').
    For any other cmd, process the cmd.
    Process command acts on current program index
    according to input cmd, modifies the program
    tree in most cases, gets new program index for
    further processing. Continue interpreter by
    getting a new cmd and interpreting the new cmd. **/

interp:-
  repeat,
  makewindow(2,7,7," COMMAND ",20,28,3,22),
  write("Type new command: "),Readkey(Key),
  removewindow,get_dbvar(progndx(Pndx)),
  process_cmd(Key,Pndx,Newpndx),
  set_progndx(Newpndx),
  chk_interp(Key),!.

chk_interp(char('q')):-
  !,bell,save("test.dba"),nl,
  makewindow(7,7,0,"",20,20,2,40),
  write("          *** goodbye ***"),nl,
  write("    *** Type any key to Exit *** "),
  readkey(_),removewindow.
```

```
/**** process_cmd ****/
/** process prog tree at pndx acc to cmd resulting
    in modified prog tree.  Newpndx will be the
    index for further processing                    **/
/** Move to next nt Newpndx to be processed **/
process_cmd(Cmd,Pndx,Newpndx):-
  is_cursor(Cmd),
  struc_move(Cmd,Pndx,Newpndx),!.

/** Ellipsis: All expanded children of Pndx in
      remains in the program tree, but aren't
      displayed.  This is done by turning display
      flag of nt off. Next prog index is the
      the same as the present                       **/
process_cmd(Cmd,Pndx,Newpndx):-
  is_ellipsis(Cmd),
  hide_children(Pndx,Newpndx),!.

/** Out_Ellipsis: Opposite of  Ellipsis.
      The display flag of nt is turned back on.
      No change in prog index                       **/
process_cmd(Cmd,Pndx,Newpndx):-
  is_out_ellipsis(Cmd),
  open_children(Pndx,Newpndx),!.

process_cmd(Cmd,Pndx,Newpndx):-
  is_save(Cmd),
  save_pgm,Newpndx=Pndx,!.

process_cmd(char('e'),Pndx,Newpndx):-
  expand(Pndx,Newpndx),!.

/** unknown command. Don't do anything **/
process_cmd(_,Pndx,Newpndx):-
  Newpndx=Pndx,!.

/**** is_cursor & others ****/
/** classify the command types **/
is_cursor(up):-!.
is_cursor(down):- !.
is_cursor(left):-!.
is_cursor(right):- !.
is_ellipsis(char('.')):-!.
is_out_ellipsis(char('o')):- !.
is_save(char('s')):-!.
```

```
/******* save_pgm *******/
save_pgm:-
  makewindow(6,7,7," *** SAVE *** ",20,10,3,60),
  write("Type File name (no extensions): "),
  readln(Fn1),
  chk_syntax(Fn1,Fn2),concat(Fn2,".pgm",Fn),
  shiftwindow(6),removewindow,
  openwrite(pgmfile,Fn),writedevice(pgmfile),
  save_all(1),
  closefile(pgmfile),
  writedevice(screen),
  makewindow(7,7,0,"",20,20,1,40),
  write("*** Type any key to continue *** "),
  readkey(_),removewindow.

/******* save_all *******/
save_all(Pndx):-
  Pndx<=0,!.
save_all(Pndx):-
  p(Pndx,const(Str),_,S1),!,
  write(Str), write(" "),
  save_all(S1).

save_all(Pndx):-
  p(Pndx,nl(Tab),_,S1),!,
  nl,write_blanks(Tab,0),
  save_all(S1).
save_all(Pndx):-
  p(Pndx,nt(_,C1,_,_,_,_),_,S1),!,
  save_all(C1),save_all(S1). /* recursive calls */

/****** write_blanks *******/
write_blanks(Tab,S):-
  S=Tab,!.
write_blanks(Tab,S):-
  S1=S+1,write(" "), write_blanks(Tab,S1).

/**** update_currRC ****/
/** update Row Col of current prog index.  **/
update_currRC(R,C):-
  del_all(curr_RC),asserta(curr_RC(R,C)).

/**** chk_dummyPar ****/
/** If the Parent has no other child other than
    the current nt prog Ndx, then Parent is dummy **/
chk_dummyPar(Ndx,Parndx):-
  get_dbvar(p(Ndx,nt(_,_,_,_,_,_),Pndx,Sndx)),
  Sndx=0,get_dbvar(p(Pndx,nt(_,Ndx,_,_,_,_),_,_)),!,
  chk_dummyPar(Pndx,Parndx).
chk_dummyPar(Ndx,Parndx):-
  Parndx=Ndx.
```

```
/**** struc_move ****/
struc_move(up,Pndx,Newpndx):-
  !,move_out(Pndx,Newpndx).
struc_move(down,Pndx,Newpndx):-
  !,move_in(Pndx,Newpndx).
struc_move(right,Pndx,Newpndx):-
  !,move_right(Pndx,Newpndx).
struc_move(left,Pndx,Newpndx):-
  !,move_left(Pndx,Newpndx).

/****** mod_col ******/
/** col modification is done only for all nt's
    in the same row as current_row.            **/
mod_col(R,C,Cd,C1):-
  get_dbvar(curr_RC(Rc,_)),Rc=R,!,C1=C-Cd.
mod_col(_,C,_,C).

/***** modify_RC ******/
/** reduces the Pndx's RC by Rd,Cd; continues
    with child and sibling link of Pndx       **/
modify_RC(Pndx,_,_):-
  Pndx<=0,!.

modify_RC(Pndx,Rd,Cd):-
  get_dbvar(p(Pndx,nt(Nt,C1,n,E,R,C),P1,S1)),
  retract_fst(p(Pndx,_,_,_)),R1=R-Rd,mod_col(R,C,Cd,C1),
  asserta(p(Pndx,nt(Nt,C1,n,E,R1,C1),P1,S1)),
  get_rightnt(Pndx,Rndx),get_fst_ntchld(Pndx,Cndx),
  modify_RC(Cndx,Rd,Cd),modify_RC(Rndx,Rd,Cd).
modify_RC(Pndx,Rd,Cd):-
  get_dbvar(p(Pndx,nt(Nt,C1,y,E,R,C),P1,S1)),
  retract_fst(p(Pndx,_,_,_)),R1=R-Rd,mod_col(R,C,Cd,C1),
  asserta(p(Pndx,nt(Nt,C1,y,E,R1,C1),P1,S1)),
  get_rightnt(Pndx,Rndx),modify_RC(Rndx,Rd,Cd).

/****** chk_uptree ******/
/** If the Plink of Pndx is not <=0
     then modify_righttree of Plink,
     else do nothing.                      **/
chk_uptree(Pndx,Rd,Cd):-
  get_dbvar(p(Pndx,_,P1,_)),P1>0,!,
  modify_righttreeRC(P1,Rd,Cd).
chk_uptree(_,_,_).

/****** modify_righttreeRC ******/
modify_righttreeRC(Pndx,Rd,Cd):-
  get_rightnt(Pndx,Rndx),
  modify_RC(Rndx,Rd,Cd),
  chk_uptree(Pndx,Rd,Cd).
```

```
/**** hide_children ****/
/** Ellipsis of nt. Turn off display flag **/
/*  modify RC's of all the nt's in the right
    tree of current Pndx.  Col change stops
    once a newline is encountered.  The diff
    is found by calculating the total R & C
    taken up due to the expansion of Pndx.    **/
hide_children(Pndx,Newpndx):-
  get_dbvar(p(Pndx,nt(Ntname,C1,n,y,R,C),P1,S1)),!,
  get_final_ntrc(Pndx,Rf,Cf),Rd=Rf-R,Cd=Cf-C,
  retract_fst(p(Pndx,_,_,_)),
  asserta(p(Pndx,nt(Ntname,C1,y,y,R,C),P1,S1)),
  modify_righttreeRC(Pndx,Rd,Cd),
  Newpndx=Pndx,disp(Pndx).
hide_children(Pndx,Newpndx):-
  !,bell,Newpndx=Pndx.

/**** open_children ****/
/** Expand ellipsis. Turn on display flag **/
open_children(Pndx,Newpndx):-
  get_dbvar(p(Pndx,nt(Ntname,C1,y,y,R,C),P1,S1)),
  retract_fst(p(Pndx,_,_,_)),
  asserta(p(Pndx,nt(Ntname,C1,n,y,R,C),P1,S1)),
  get_final_ntrc(Pndx,Rf,Cf),Rd=R-Rf,Cd=C-Cf,
  modify_righttreeRC(Pndx,Rd,Cd),
  Newpndx=Pndx,disp(Pndx).
open_children(Pndx,Newpndx):-
  !,bell,Newpndx=Pndx.

/**** get_fst_ntchld ****/
/** if child link of nt points to another
      nt then Cntndx=Clink.
      Otherwise, get the child index, traverse
      right using sibling link till a nt
      index Cntndx is obtained.                **/
get_fst_ntchld(Pndx,Cntndx):-
  get_dbvar(p(Pndx,nt(_,Cntndx,n,_,_,_),_,_)),
  get_dbvar(p(Cntndx,nt(_,_,_,_,_,_),_,_)),!.
/** no nt child.  get_rightnt returns index <=0 **/
get_fst_ntchld(Pndx,Cntndx):-
  get_dbvar(p(Pndx,nt(_,Constndx,n,_,_,_),_,_)),
  get_rightnt(Constndx,Cntndx).

/**** get_last_ntchld ****/
/** Def: Lntchld is the last nt child of Pndx.
         if Pndx doesn't have ntchild, then
         Lntchld is set to 0                     **/
get_last_ntchld(Pndx,Lntchld):-
  p(Lntchld,nt(_,_,_,_,_,_),Pndx,_),
  get_rightnt(Lntchld,Rndx),Rndx<=0,!.
get_last_ntchld(_,Lntchld):-
  !,Lntchld=0.
```

```
/****** get_final_ntrc ******/
/** should be only used with expanded nt's
    whose child link is open                **/
get_final_ntrc(Ntndx,Rf,Cf):-
  get_last_ntchld(Ntndx,Lndx),Lndx<=0,!,
  /** no nt child **/
  get_dbvar(p(Ntndx,nt(_,Cl,n,_,R,C),_,_)),
  restRC(Cl,R,C,Rf,Cf).
get_final_ntrc(Ntndx,Rf,Cf):-
  get_last_ntchld(Ntndx,Lndx),
  get_dbvar(p(Lndx,nt(_,_,y,_,Rs,Cs),_,_)),!,
  restRC(Lndx,Rs,Cs,Rf,Cf).
get_final_ntrc(Ntndx,Rf,Cf):-
  get_last_ntchld(Ntndx,Lndx),!,
  /** nt whose child link to be followed **/
  get_final_ntrc(Lndx,Rf1,Cf1),
  get_dbvar(p(Lndx,nt(_,_,n,_,_,_),_,S1)),
  restRC(S1,Rf1,Cf1,Rf,Cf).

/****** restRC ******/
/** used to find get_final_ntrc
    Rf and Cf are row and col of parent node
    of S index after its expansion          **/
restRC(S,Rs,Cs,Rf,Cf):-
  S<=0,!,Rf=Rs,Cf=Cs. /* end of sibling link */
restRC(S,Rs,_,Rf,Cf):-
  get_dbvar(p(S,nl(Tab),_,S1)),!,Rs1=Rs+1,Cs1=Tab,
  restRC(S1,Rs1,Cs1,Rf,Cf).
restRC(S,Rs,Cs,Rf,Cf):-
  /** either displayable nt or const **/
  !,getlen(S,Len),Cs1=Cs+Len+1,
  get_dbvar(p(S,_,_,S1)),restRC(S1,Rs,Cs1,Rf,Cf).

/**** get_leftnt ****/
/** Get parent and its first nt chlid.
      Chk_rightnt takes this nt child, follows
      sibling link to get left nt.
      No left nt means Lpndx=0              */
get_leftnt(Pndx,Lpndx):-
  get_dbvar(p(Pndx,_,Plink,_)),Plink<=0,!,
  Lpndx=0. /* root node */
get_leftnt(Pndx,Lpndx):-
  get_dbvar(p(Pndx,_,Plink,_)),
  get_fst_ntchld(Plink,Chldndx),
  chk_rightnt(Chldndx,Pndx,Lpndx).
```

```
/**** chk_rightnt ****/
/** def: Lpndx is the right index of Chldndx
        and left index of Pndx                          **/
chk_rightnt(Chldndx,_,Lpndx):-
  Chldndx<=0,!,Lpndx=0. /* no left sibling */
chk_rightnt(Chldndx,Pndx,Lpndx):-
  Chldndx=Pndx,!,Lpndx=0. /* no left sibling */
chk_rightnt(Chldndx,Pndx,Lpndx):-
  get_rightnt(Chldndx,Rndx),
  Rndx=Pndx,!,Lpndx=Chldndx.
chk_rightnt(Chldndx,Pndx,Lpndx):-
  get_rightnt(Chldndx,Rndx),!,
  chk_rightnt(Rndx,Pndx,Lpndx).

/**** get_rightnt ****/
/** Def: Rpndx is right nt of Pndx. If Pndx
        doesn't have right nt, then Rpndx <=0  **/
get_rightnt(Pndx,Rpndx):-
  get_dbvar(p(Pndx,_,_,Rpndx)),
  get_dbvar(p(Rpndx,nt(_,_,_,_,_,_),_,_)),!.
get_rightnt(Pndx,Rpndx):-
  get_dbvar(p(Pndx,_,_,Rpndx)),Rpndx<=0,!.
get_rightnt(Pndx,Rpndx):-
  get_dbvar(p(Pndx,_,_,Slink)),!,get_rightnt(Slink,Rpndx).

/**** chk_display ****/
/** If the Pndx is within screen, only highlight
    need to be changed.                                 **/
chk_display(Pndx):-
  get_dbvar(curr_RC(Rc,Cc)),get_dbvar(maxline(L)),
  get_dbvar(diff(Old_diff)),
  Rcl=(Rc-Old_diff),Rcl>=0,Rcl<L,
  !,   /* node in same screen */
  get_fst_str(Pndx,Str),update_curr_str(Str),
  str_len(Str,Len),
  field_attr(Rcl,Cc,Len,28),
  field_str(Rcl,Cc,Len,Str),!.
chk_display(Pndx):-
  !,disp(Pndx).

/**** chk_in_disp ****/
/** If prev index and current prog index are
      the same, do nothing.  Otherwise remove
      the highlight from prev index, check
      if the new index is within screen, dislay
      and highlight the newpndx contents            **/
chk_in_disp(Pndx,Newpndx):-
  Pndx=Newpndx,!,bell.
```

```
chk_in_disp(_,Newpndx):-
  get_dbvar(diff(Diff)),
  get_dbvar(curr_RC(Rc,Cc)),Rc1=Rc-Diff,
  get_dbvar(curr_str(Str)),cursor(Rc1,Cc),write(Str),
  getRC(Newpndx,R1,C1),
  update_currRC(R1,C1),chk_display(Newpndx).

/**** move_in ****/
/** If not expanded(i.e.,Clink<=0), or not to
      be displayed because of Ellipsis, return
      New prog index the current one and do
      nothing. Otherwise, get first nt and
      highlight the new index.              **/
move_in(Pndx,Newpndx):-
  get_dbvar(p(Pndx,nt(_,Clink,_,_,_,_),_,_)),Clink<=0,!,
  bell,Newpndx=Pndx. /* no child. not expanded */
move_in(Pndx,Newpndx):-
  get_dbvar(p(Pndx,nt(_,_,y,_,_,_),_,_)),!,
  bell,Newpndx=Pndx. /* Ellipsed nt */
move_in(Pndx,Newpndx):-
  get_dbvar(p(Pndx,nt(_,Clink,_,_,_,_),_,_)),
  get_dbvar(p(Clink,const(_),_,_)),
  get_fst_ntchld(Pndx,Newpndx1),
  Newpndx1<=0,!, /* no nt child */
  bell,Newpndx=Pndx.
move_in(Pndx,Newpndx):-
  get_dbvar(p(Pndx,nt(_,Clink,_,_,_,_),_,_)),
  get_dbvar(p(Clink,const(_),_,_)),!,
  get_fst_ntchld(Pndx,Newpndx),
  chk_in_disp(Pndx,Newpndx).
/** If first child is nt, move another level in **/
move_in(Pndx,Newpndx):-
  get_dbvar(p(Pndx,nt(_,Clink,_,_,_,_),_,_)),
  get_dbvar(p(Clink,nt(_,_,_,_,_,_),_,_)),!,
  move_in(Clink,Newpndx).

/**** move_out ****/
/** Def: Newpndx is parent nt of Pndx and is
         currently highlighted on the screen.
         If the Pndx is the first child of its
         parent, move to grand parent and so on **/
move_out(Pndx,Newpndx):-
  get_dbvar(p(Pndx,_,Plink,_)),Plink<=0,!,
  bell,Newpndx=Pndx. /* root node */
move_out(Pndx,Newpndx):-
  get_dbvar(p(Pndx,_,Newpndx1,_)), /* Pndx RC = Newpndx1 RC */
  get_dbvar(p(Newpndx1,nt(_,Pndx,_,_,_,_),_,_)),!,
  move_out(Newpndx1,Newpndx).
move_out(Pndx,Newpndx):-
  get_dbvar(p(Pndx,_,Newpndx,_)),!,
  chk_in_disp(Pndx,Newpndx).
```

```
/**** move_left ****/
/** Def: Newpndx is the new nt prog index
         which is left nt of Pndx & is currently
         highlighted on the screen.              **/
move_left(Pndx,Newpndx):-
  get_leftnt(Pndx,Newpndx1),
  Newpndx1<=0,!,bell,Newpndx=Pndx.
move_left(Pndx,Newpndx):-
  get_leftnt(Pndx,Newpndx),
  chk_in_disp(Pndx,Newpndx).

/**** move_right ****/
/** Def: Newpndx is right nt of Pndx which is
         currently highlighted on the screen  **/
move_right(Pndx,Newpndx):-
  get_dbvar(p(Pndx,_,_,Newpndx1)),Newpndx1<=0,!,
  bell,Newpndx=Pndx.
move_right(Pndx,Newpndx):-
  get_rightnt(Pndx,Newpndx),!,
  chk_in_disp(Pndx,Newpndx).

/**** update_curr_str ****/
/** curr_str is the one highlighted on the screen **/
update_curr_str(Str):-
  del_all(curr_str),asserta(curr_str(Str)).

/**** get_fst_str ****/
/** def: Str is the displayable string of Pndx **/
get_fst_str(Pndx,Str):-
  get_dbvar(p(Pndx,nt(Ntname,_,y,_,_,_),_,_)),!,
  brace_nt(Ntname,Str).
get_fst_str(Pndx,Str):-
  get_dbvar(p(Pndx,nt(_,Chldndx,_,_,_,_),_,_)),
  get_dbvar(p(Chldndx,nt(Ntname,_,y,_,_,_),_,_)),!,
  brace_nt(Ntname,Str).
get_fst_str(Pndx,Str):-
  get_dbvar(p(Pndx,nt(_,Chldndx,_,_,_,_),_,_)),
  get_dbvar(p(Chldndx,const(Str),_,_)),!.
get_fst_str(Pndx,Str):-
  get_dbvar(p(Pndx,nt(_,Chldndx,_,_,_,_),_,_)),
  get_dbvar(p(Chldndx,nt(_,_,_,_,_,_),_,_)),!,
  get_fst_str(Chldndx,Str).

/*-------------------------------------------------------*/
/*                  --- menu system ---                  */
/*-------------------------------------------------------*/
predicates
  maxlen(symlist,integer,integer)
    /* maxlen(in:SymbolList,in:InitCol,out:WidestCol) */

  listlen(symlist,integer)
    /* listlen(in:SymbolList,out:Numofitems) */
```

```
  writelist(integer,integer,symlist)
    /* writelist(in:StartRow,in:StartCol,in:SymbolList) */

  menu(integer,integer,string,symlist,integer)
    /* menu(in:Topleftrow,in:Topleftcol,
            in:Windowtext,in:Listofchoices,out:Choice) */

  menu1(integer,symlist,integer,integer,integer)
    /* menu1(in:InitRow,in:SymbolList,
             in:MaxRows,in:MaxCol,out:Choice) */

  menu2(integer,symlist,integer,integer,integer,key)
    /* menu1(in:InitRow,in:SymbolList,in:MaxRows,
             in:MaxCol,out:Choice,in:Key)         */

/*-------------- clauses for menu system -----------------*/
clauses

/******* maxlen ********/
maxlen([H|T],Max,Max1):-
  str_len(H,Len),Len>Max,!,
  maxlen(T,Len,Max1).
maxlen([_|T],Max,Max1):-
  maxlen(T,Max,Max1).
maxlen([],Len,Len).

/******* listlen *******/
listlen([],0).
listlen([_|T],N):-
  listlen(T,X),N=X+1.

/******* writelist *******/
writelist(_,_,[]).
writelist(IRow,ICol,[H|T]):-
  field_str(IRow,0,ICol,H),IRow1=IRow+1,
  writelist(IRow1,ICol,T).

/******* menu *******/
menu(IRow,ICol,Txt,SymList,Choice):-
  maxlen(SymList,0,FCol),
  listlen(SymList,Len),FRow=Len,Len>0,
  MRow=FRow+2,MCol=FCol+10, /* height & width of popup window */
  IRow1=IRow,ICol1=ICol, /* adjust popup window for prog window coords */
  PopR=IRow1+MRow,
  makewindow(5,112,0,"",PopR,ICol1,2,MCol), /* info menu for pop up menu */
  write(" Arrows:Choose"),nl,
  write(" Cr    :Select"),
  makewindow(4,7,7,Txt,IRow1,ICol1,MRow,MCol),
  FCol1=FCol,
  writelist(0,FCol1,Symlist),cursor(0,0),
  menu1(0,SymList,FRow,FCol,Ch),
```

```
  Choice=Ch+1,
  removewindow, /* pop up menu */
  removewindow, /* info menu for pop up menu */
  shiftwindow(99), /* background masking window */
  shiftwindow(21), /* command info menu */
  shiftwindow(1). /* shift back to program disp window */

/******* menu1 *******/
menu1(IRow,SymList,FRow,FCol,Choice):-
  field_attr(IRow,0,FCol,112),
  cursor(IRow,0),readkey(Key),
  menu2(IRow,SymList,FRow,FCol,Choice,Key).

/******* menu2 *******/
menu2(IRow,_,_,_,Ch,cr):-
  !,Ch=IRow.
menu2(IRow,SymList,FRow,FCol,Choice,up):-
  IRow>0,!,field_attr(IRow,0,FCol,7),
  IRow1=IRow-1,
  menu1(IRow1,SymList,FRow,FCol,Choice).
menu2(IRow,SymList,FRow,FCol,Choice,down):-
  IRow<FRow-1,!,field_attr(IRow,0,FCol,7),
  IRow1=IRow+1,
  menu1(IRow1,SymList,FRow,FCol,Choice).
menu2(IRow,SymList,FRow,FCol,Choice,_):-
  /* no action for all other keys */
  menu1(IRow,SymList,FRow,FCol,Choice).

/*-------------------------------------------------------*/
/*                  --- expansion ---                    */
/*-------------------------------------------------------*/
predicates
  get_next_num(integer)
    /* get_next_number(out: NextPndx) */

  expand_next(integer,integer)
    /* expand_next(in:Ndx,in:Newndx) */

  get_chldnode(integer,integer)
    /* get_chldnode(in:Ndx,out:Next) */

  get_rnode(integer,integer)
    /* get_rnode(in:Ndx,out:Next) */

  chk_expand(integer,integer,integer).
    /* chk_expand(in:Ndx,in:Next,out:Newndx) */

  link_rhs(integer,integer,integer)
    /* link_rhs(in:Tndx,in:Num,in:ParNdx) */
    /* links rhs of nt into the prog tree. */
```

```
  semact(integer,integer,integer)
    /* semact(in:Tndx,in:Num,in:Ndx) */

chk_more_choose(integer,integer,integer)
    /* chk_more(in:Tndx,in:Ndx,out:Newndx) */

  idget(symbol,integer,integer)
    /* idget(in:Idtype,in:Num,in:Parndx) */

  chk_getid(symbol,string)
    /* chk_getid(in:Idtype,in:Val) */

  chk_symtab(symbol,string)
    /* chk_symtab(in:Idtype,out:Val) */

  chk_uniqueId(symbol,string,string)
    /* chk_uniqueId(in:Idtype,in:Val2,out:Val) */

  choice_node(symbol,integer,integer)
    /* choice_node(in:Choicetype,in:Num,in:Parndx) */

  get_choice_str(integer,integer,symlist,symbol)
    /* get_choice_str(in:Choice,in:Count,
                      in:Choicelist,out:ChStr) */

  moresa(symbol,integer,integer)
    /* moresa(in:Saparm,in:Num,in:Parndx) */

/*--------------- clauses for expand ------------------*/
clauses

/****** get_next_num ******/
get_next_num(Num):-
  retract(curr_num(Num1)),!,Num = Num1+1,
  asserta(curr_num(Num)).

/****** expand ******/
expand(Ndx,Newndx):-
  /* can't expand constant */
  p(Ndx,const(_),_,_),!,bell,Newndx=Ndx.

expand(Ndx,Newndx):-
  /* can't expand nl */
  p(Ndx,nl(_),_,_),!,bell,Newndx=Ndx.

expand(Ndx,Newndx):-
  /* nt already expanded */
  p(Ndx,nt(_,_,_,y,_,_),_,_),!,bell,
  Newndx=Ndx.
```

```
expand(Ndx,Newndx):-
  get_dbvar(p(Ndx,nt(Ntname,_,_,n,R,C),P1,S1)),
  get_dbvar(d(Ntname,Tndx)),
  get_dbvar(t(Tndx,sa(_,_),0)),!,
  get_next_num(Num),
  retract_fst(p(Ndx,_,_,_)),
  assertz(p(Ndx,nt(Ntname,Num,n,y,R,C),P1,S1)),
  semact(Tndx,Num,Ndx),
  chk_more_choose(Tndx,Ndx,Newndx).

expand(Ndx,Newndx):-
  /* Note: Db Num will be Current max+1 before predicate
           link_rhs is called because to get the Child
           link of a nt, the next available number is to
           be used.                                        */
  p(Ndx,nt(Ntname,_,_,n,R,C),P1,S1),!,
  get_dbvar(d(Ntname,Tndx)),get_next_num(Num),
  retract_fst(p(Ndx,_,_,_)),
  assertz(p(Ndx,nt(Ntname,Num,n,y,R,C),P1,S1)),
  link_rhs(Tndx,Num,Ndx),
  get_final_ntrc(Ndx,Rf,Cf),Rd=R-Rf,Cd=C-Cf,
  modify_righttreeRC(Ndx,Rd,Cd),
  expand_next(Ndx,Next),
  chk_expand(Ndx,Next,Newndx).

/***** chk_more *******/
chk_more_choose(Tndx,Ndx,Newndx):-
  t(Tndx,sa(getid,_),_),!,
  expand_next(Ndx,Next),
  chk_expand(Ndx,Next,Newndx).

chk_more_choose(Tndx,Ndx,Newndx):-
  t(Tndx,sa(choose,_),_),!,
  expand_next(Ndx,Next),Next>0,
  expand(Next,Newndx). /* indirect recursion */

chk_more_choose(Tndx,Ndx,Newndx):-
  t(Tndx,sa(more,_),_),!,
  get_leftnt(Ndx,Lndx),
  get_dbvar(p(Lndx,nt(_,_,y,n,_,_),_,_)),
  expand(Lndx,Newndx). /* indirect recursion */

/***** chk_expand ******/
chk_expand(_,Next,Newndx):-
  Next>0,!,Newndx=Next,disp(Newndx).

chk_expand(Ndx,_,Newndx):-
  bell,Newndx=Ndx,disp(Newndx),!.
```

```
/******* get_chldnode ********/
get_chldnode(Ndx,Next):-
  Ndx<=0,!,Next=0.

get_chldnode(Ndx,Next):-
  get_fst_ntchld(Ndx,Next),
  p(Next,nt(_,_,y,n,_,_),Ndx,_),!.

get_chldnode(Ndx,Next):-
  get_fst_ntchld(Ndx,Next1),
  get_rnode(Next1,Next).

/******* get_rnode **********/
get_rnode(Ndx,Next):-
  Ndx<=0,!,Next=0.

get_rnode(Ndx,Next):-
  get_rightnt(Ndx,Next),
  p(Next,nt(_,_,y,n,_,_),_,_),!.

get_rnode(Ndx,Next):-
  get_rightnt(Ndx,Next1),
  get_rnode(Next1,Next). /* recursive call */

/******* expand_next *******/
expand_next(Ndx,Next):-
  get_chldnode(Ndx,Next),Next>0,!.

expand_next(Ndx,Next):-
  get_rnode(Ndx,Next),Next>0,!.

expand_next(Ndx,Next):-
  get_dbvar(p(Ndx,nt(_,_,_,_,_,_),P1,_)),
  P1>0,expand_next(P1,Next). /* recursive call */

expand_next(_,0). /* reached the root */

/***** link_rhs *****/
link_rhs(Tndx,Num1,Ndx):-
  Tndx<=0, /* end of rhs tokens */
  /* adjust Db curr_num */
  Num = Num1-1,retract(curr_num(Num1)),!,
  asserta(curr_num(Num)),
  /* adjust the sibling link of prev p node to 0 */
  retract_fst(p(Num,Node,Ndx,Num1)),
  assertz(p(Num,Node,Ndx,0)).
/* The terminating case of Ts=0 is handled
   seperately by the first rule above      */
link_rhs(Tndx,Num,Ndx):-
  t(Tndx,const(Str),Ts),!,
  get_next_num(Num1),
  assertz(p(Num,const(Str),Ndx,Num1)),
```

```
  link_rhs(Ts,Numl,NdX). /* recursive call */

link_rhs(Tndx,Num,Ndx):-
  t(Tndx,nl(Tab),Ts),!,
  get_next_num(Numl),
  assertz(p(Num,nl(Tab),Ndx,Numl)),
  link_rhs(Ts,Numl,NdX). /* recursive call */

link_rhs(Tndx,Num,Ndx):-
  t(Tndx,nt(Ntname),Ts),!,
  get_next_num(Numl),
  assertz(p(Num,nt(Ntname,-1,y,n,-1,-1),Ndx,Numl)),
  getlRC(Num,R,C),
  retract_fst(p(Num,nt(Ntname,-1,y,n,_,_),Ndx,Numl)),
  assertz(p(Num,nt(Ntname,-1,y,n,R,C),Ndx,Numl)),
  link_rhs(Ts,Numl,Ndx). /* recursive call */

/****** semact ******/
/* Note: The semantic action token will not have any
         sibling link. i.e. the definition nt token
         which leads to SA token will only have SA
         token as its rhs                              */
semact(Tndx,Num,Parndx):-
  t(Tndx,sa(getid,Idtype),0),!,
  idget(Idtype,Num,Parndx).

semact(Tndx,Num,Parndx):-
  t(Tndx,sa(choose,Choicetype),0),!,
  choice_node(Choicetype,Num,Parndx).

semact(Tndx,Num,Parndx):-
  t(Tndx,sa(more,Moreparm),0),!,
  moresa(Moreparm,Num,Parndx).

/******* moresa *******/
moresa(MoreParm,Num,Parndx):-
  retract_fst(p(Ndx,Node,Pl,Parndx)),
  assertz(p(Ndx,Node,Pl,Num)),
  retract_fst(p(Parndx,nt(Pname,_,n,y,Rm,Cm),Pm,Sm)),
  get_next_num(Numl),
  assertz(p(Num,nt(Moreparm,-1,y,n,Rm,Cm),Pm,Numl)),
  assertz(p(Numl,nl(Cm),Pm,Parndx)),
  Rml=Rm+1,
  assertz(p(Parndx,nt(Pname,-1,y,n,Rml,Cm),Pm,Sm)),
  modify_righttreeRC(Parndx,-1,0). /* propagate RC change */

/******* get_choice_Str *******/
/* Assumption: Choice <= total elements in the list */
get_choice_str(Choice,Choice,[H|_],H):-!.
get_choice_str(Choice,Count,[_|T],ChStr):-
  Countl=Count+1,
  get_choice_str(Choice,Countl,T,ChStr). /* recursive call */
```

```
/******* choice_node *******/
choice_Node(Choicety,Num,Parndx):-
  c(Choicety,Listchoice),!,
  str_symbol(Txt,Choicety),
  concat(" ",Txt,Txt1),concat(Txt1," ",Txt2),
  get_dbvar(p(Parndx,nt(Ntname,Num,_,_,R,C),_,_)),
  get_dbvar(diff(D)),R1=R-D, /* menu window row */
  str_len(Ntname,Len),C1=C+Len+8,
  menu(R1,C1,Txt2,Listchoice,Choice),
  get_choice_str(Choice,1,Listchoice,Str),
  assertz(p(Num,nt(Str,-1,y,n,R,C),Parndx,0)).

/******** idget ********/
idget(Idtype,Pndx,Parndx):-
  chk_symtab(Idtype,Val),
  assertz(p(Pndx,const(Val),Parndx,0)),
  get_dbvar(p(Parndx,nt(_,_,_,_,R,C),_,_)),
  get_final_ntrc(Parndx,Rf,Cf),Rd=R-Rf,Cd=C-Cf,
  modify_righttreeRC(Parndx,Rd,Cd),
  chk_getid(Idtype,Val).

/******* chk_getid ********/
chk_getid(Idtype,Val):-
  retract(p(Parndx,nt(Idtype,_,y,n,R,C),P1,S1)), /* has backtrack points */
  get_next_num(Num),
  assertz(p(Parndx,nt(Idtype,Num,n,y,R,C),P1,S1)),
  assertz(p(Num,const(Val),Parndx,0)),
  get_final_ntrc(Parndx,Rf,Cf),Rd=R-Rf,Cd=C-Cf,
  modify_righttreeRC(Parndx,Rd,Cd),
  fail. /* force backtracking */

chk_getid(_,_).

/******* chk_symtab *******/
chk_symtab(Idtype,Val):-
  s(Idtype,Val),!.
chk_symtab(Idtype,Val):-
  bell,makewindow(3,7,7," *** GET ID *** ",19,10,4,60),
  write("Type Identifier for ",Idtype," and enter: "),
  readln(Val1),chk_syntax(Val1,Val2),
  chk_uniqueId(Idtype,Val2,Val),
  asserta(s(Idtype,Val)),removewindow.

/****** chk_syntax *******/
chk_syntax(Val1,Val2):-
  isname(Val1),!,Val2=Val1.
chk_syntax(_,Val2):-
  bell,write("*** Syntax Error ***"),nl,
  write("Try Again: "),readln(Val3),
  chk_syntax(Val3,Val2). /* recursive call */
```

```
/****** chk_uniqueId ******/
chk_uniqueId(Idtype,Val2,Val):-
  s(_,Val2),!,bell,
  write("The identfier already exists.  "),
  write("Try again: "),
  readln(Val4),chk_syntax(Val4,Val5),
  chk_uniqueId(Idtype,Val5,Val).
chk_uniqueId(_,Val2,Val):-
  Val=Val2.
```

INTERACTIVE TECHNIQUES FOR A PROGRAM GENERATOR USING PROLOG

by

K. S. Venkatesh

B.S., Mechanical Engg., Bangalore University, India, 1979

M.S., Mechanical Engg., Kansas State University, 1984

AN ABSTRACT OF A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

Automatic program generators provide an environment for assisting program generation typically in a target language. Such program generators use a knowledge base of generic program algorithms on data structures stored as program plans. In a very limited sense, structured editors incorporating syntactic knowledge of target language, provide such an environment. The programming activity involves knowledge of syntax, semantics, and pragmatics. This thesis includes a categoric survey of various program generators according to the internal knowledge representation schemes.

This thesis reports on a structured representation of programming plans in a logic programming environment. The research includes an investigation of logic programming paradigms for program generation, representation of program plans as database facts, and the design of user interfaces. The representation of program focuses on issues such as efficiency and manipulative power. The proposed representation is compared with two others: functional style in Prolog using lists, and list representation in an imperative language.

A prototype program generation system was implemented in Turbo Prolog to experiment with some of the issues in program generation. A description of the implementation is included.